TAMPEREEN TEKNILLINEN YLIOPISTO

ANTTI LEHTINEN

MALLI-NÄKYMÄ-ARKKITEHTUURIN SOVELTAMINEN WPF-TEKNOLOGIALLA

Diplomityö

Tarkastaja: professori Kai Koskimies
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan tiedekunta-neuvoston kokouksessa
8. kesäkuuta 2011

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
LEHTINEN, ANTTI: **Malli-näkymä-arkkitehtuurin soveltaminen WPF-teknologialla.**
Diplomityö, 76 sivua, 24 liitesivua
Toukokuu 2012
Pääaine: Ohjelmistotuotanto
Tarkastaja: professori Kai Koskimies
Avainsanat: malli-näkymä-arkkitehtuuri, WPF, MVVM-suunnittelumalli

Malli-näkymä-arkkitehtuurit tarjoavat työkaluja ja suunnitteluratkaisuja käyttöliittymäsovelluksien toteutukseen. Malli-näkymä-arkkitehtuurien tehtävänä on parantaa käyttöliittymäsovelluksien modulaarisuutta, toteutettavuutta, ylläpidettävyyttä sekä testattavuutta. Tärkein malli-näkymä-arkkitehtuureista saavutettava hyöty on näkymän erottaminen sovelluslogiikasta. Tunnetuimpia malli-näkymä-arkkitehtuureja ovat MVC ja MVP.

Tässä työssä esitellään tunnetuimpien malli-näkymä-arkkitehtuurien rakenteet ja tehtävät. Työn alussa huomataan, että MVC- ja MVP-arkkitehtuurimallien käytännön soveltaminen on vaikeaa. Tämän diplomityön tarkoituksena on selvittää, kuinka kirjallisuudessa hyväksi todettu uusi malli-näkymä-arkkitehtuuri MVVM ja käyttöliittymäkirjasto WPF ratkaisevat todellisuudessa ongelmia, jotka liittyvät käyttöliittymäsovelluksien toteuttamiseen.

Perehtyminen MVVM-suunnittelumallin mukaisen järjestelmän toteutukseen aloitetaan esimerkkisovelluksen avulla. Esimerkkisovellus havainnollistaa WPF-teknologian tärkeimmät ominaisuudet liittyen MVVM-suunnittelumallin toteutukseen. Esimerkkisovellus on toteutettu kirjallisuudessa esitettyjen ratkaisujen mukaisesti.

Havaintoja ja kokemuksia MVVM-suunnittelumallin käytännön soveltamisesta saatiin Atostekin projektista. Projektissa toteutettiin asiakkaan vaatimia mittausjärjestelmä hyödyntämällä MVVM-suunnittelumallia ja WPF-teknologiaa. Projektissa saatuja havaintoja käsitellään tässä työssä. Projektissa sovellettua MVVM-suunnittelumallia arvioitiin modulaarisuuden, suorituskyvyn, uudelleenkäytettävyyden ja opittavuuden näkökulmista.

Tässä työssä saatujen havaintojen perusteella MVVM-suunnittelumalli soveltuu hyvin käyttöliittymäsovelluksen arkkitehtuuriksi käytännön ohjelmistoprojektissa. MVVM-suunnittelumalli eroaa muista malli-näkymä-arkkitehtuureista, koska se tarjoaa myös matalan tason käytännön toteutusratkaisuja hyödyntäen WPF-teknologiaa. Kirjallisuudessa ja myös tässä työssä saatujen tuloksien perusteella MVVM-suunnittelumalli ja WPF-teknologia mahdollistavat nykyaikaisten ohjelmointityökalujen ja käyttöliittymäkirjastojen hyödyntämisen sekä tehokkaan tavan toteuttaa modulaarisia, uudelleenkäytettäviä, ylläpidettäviä ja opittavia käyttöliittymäsovelluksia.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
LEHTINEN, ANTTI: Implementing Model-View Architecture in WPF technology
Master of Science Thesis, 76 pages, 24 Appendix pages
May 2012
Major: Software Engineering
Examiner: Professor Kai Koskimies
Keywords: model-view architecture, WPF, MVVM pattern

Model-view architectures provide tools and design solutions for implementation of UI applications. Purpose of the model-view architecture is to improve modularity, feasibility, maintainability and testability of applications. Main achievement of the model-view architecture is to manage separation between data, behavior, and presentation. Two of the moset common model-view architectures today are MVC and MVP.

This thesis presents structures and functions of these model-view architectures. This thesis begins by introducting a discovery that the MVC and the MVP architectures are difficult to apply in practice. Purpose of this thesis is to examine how a new model-view architecture MVVM and WPF technology solves the problems related to the implementation of UI applications in practice.

Implementation of the MVVM pattern is presented through an example application. The example application demonstrates the most important features of the WPF technology related to the MVVM pattern. The example application is implemented with MVVM pattern concepts outlined in the literature.

The MVVM pattern was applied in practice during Atostek's software project. A measurement application for a customer was implemented by using the MVVM pattern and the WPF technology. Discoveries and experiences from the project are discussed in this thesis. MVVM pattern applied in the project was evaluated by modularity, performance, reusability and learnability points of view.

An experience of this thesis shows that the MVVM pattern provides benefits when it is used to create architecture for a UI application. MVVM-pattern is different from other model-view architectures, because it offers low-level implementation solutions using WPF technology. In the literature and also in practice MVVM-pattern and WPF-technology make it possible to use modern programming tools and user interface libraries to develope modular, reusable, maintainable and easily learned UI applications.

ALKUSANAT

Tämän diplomityönaiheen on tarjonnut ja rahoittanut Atostek Oy.

Haluan kiittää työn tarkastajaa professori Kai Koskimiestä ja työn ohjaajaa Miika Parviota. Kiittäisin myös Atostek Oy:n kaikkia työntekijöitä hyvän ja opettavaisen työympäristön tarjoamisesta.

Erityiskiitokset isälle ja äidille kaikista neuvoista, joita olen elämäni aikana teiltä saanut. Teidän tuki ja kannustus on vienyt minua eteenpäin omalla opiskelu-urallani. Lopuksi lämmin kiitos avopuolisolleni Leenalle kaikesta tuesta jota olet minulle antanut.

Tampereella 15.4.2012

Antti Lehtinen

SISÄLLYS

1 JOHDANTO

Arkkitehtuurilla on suuri merkitys käyttöliittymäsovelluksien toteutuksessa. Arkkitehtuurin tarjoama modulaarisuus on tärkeää, jotta järjestelmän rakenne pysyy selkeänä. Selkeän rakenteen ja vastuunjaon ansiosta järjestelmän testaus, ylläpito ja uusien ominaisuuksien lisääminen helpottuu. Käyttöliittymäsovelluksissa arkkitehtuurin on mahdollistettava tiedon välitys käyttäjän ja tietomallin välillä.

Ensimmäisistä käyttöliittymäsovelluksista alkaen malli-näkymä-arkkitehtuurit ovat tarjonneet työkaluja ja suunnitteluratkaisuja käyttöliittymäsovelluksien kehittäjille. Tunnetuin malli-näkymä-arkkitehtuuri on MVC (eng. Model-View-Controller). Vanhojen malli-näkymä-arkkitehtuurien ongelmana on, että niiden mukaisen käyttöliittymäsovelluksen toteuttaminen nykyaikaisilla ohjelmointiteknologioilla on mahdotonta. Tämän lisäksi korkean tason arkkitehtuurimallien ongelmana on arkkitehtuurimallin käytännön toteuttaminen. Korkean tason arkkitehtuurimallit ovat yleiskäyttöisiä ja teoriassa lupaavat ratkaista monia ongelmia liittyen järjestelmien toteuttamiseen. Kirjallisuudessa luvataan arkkitehtuurien parantavan modulaarisuutta, ylläpidettävyyttä, suorituskykyä, testattavuutta ja muutosten hallintaa. Arkkitehtuurimallien mahdollistamat hyödyt voidaan kuitenkin menettää epäonnistuneen toteutuksen seurauksena, koska korkean tason kuvaus ei tarjoa käytännön ratkaisuja toteuttajalle. Korkean tason arkkitehtuurimallin soveltamisesta saavutetut hyödyt riippuvat enemmän yksilön taidoista kuin käytettävästä arkkitehtuurimallista. Korkean tason arkkitehtuurimalleja soveltaville järjestelmille on tyypillistä, että järjestelmät eivät vastaa täysin alkuperäistä arkkitehtuurimallia, eivätkä ne ole keskenään yhdenmukaisesti toteutettuja. Tämä on seurausta siitä, että korkean tason arkkitehtuurimallien toteuttamiselle ei tarjota käytännön toteutusratkaisuja ja työkaluja siihen, kuinka arkkitehtuurimalli käytännössä toteutetaan niin, että sen hyödyt voidaan saavuttaa.

Microsoftin *Windows Forms*-käyttöliittymäkirjasto on yleisesti käytössä ollut toteutusteknologia graafisten käyttöliittymäsovelluksien toteuttamiselle. Microsoft on jatkanut kehitystä ja julkaissut uuden käyttöliittymäkirjaston nimeltään Windows Presentation Foundation – WPF. WPF-teknologia tarjoaa nykyaikaisia menetelmiä ja työkaluja käyttöliittymäsovelluksien toteuttamiselle. Käyttöliittymäkirjaston lisäksi WPF-teknologian kehityksessä on otettu huomioon toteutettavan järjestelmän arkkitehtuuriratkaisuja. WPF-teknologia on toteutettu tukemaan Microsoftin kehittämään MVVM-suunnittelumallia, joka on käyttöliittymäsovelluksiin toteutettava malli-näkymä-arkkitehtuuri. MVVM eroaa muista korkean tason malli-näkymä-arkkitehtuureista siten, että se ottaa kantaa myös matalantason käytännön ratkaisuihin. Näin ollen MVVM tar-

joaa järjestelmän kehittäjälle paremmat työkalut oikeaoppisen käyttöliittymäsovelluksen toteuttamiselle.

Tämän diplomityön tarkoituksena on selvittää, kuinka kirjallisuudessa hyväksi todettu malli-näkymä-arkkitehtuuri MVVM ja käyttöliittymäkirjasto WPF ratkaisevat todellisuudessa ongelmia, jotka liittyvät käyttöliittymäsovelluksien toteuttamiseen.

MVVM-suunnittelumallin toteutukseen WPF-teknologialla perehdytään esimerkkisovelluksen avulla. MVVM-suunnittelumallin sovellettavuutta ohjelmistoprojektiin arvioidaan perustuen kokemuksiin, joita saatiin hyödyntämällä MVVM-suunnittelumallia Atostekin projektissa. Projektin aikana saatuja havaintoja käydään läpi esittelemällä käyttöliittymäsovelluksen toiminnallisuuksia sovellettuna MVVM-suunnittelumallin mukaiseksi. Käsiteltäviä toiminnallisuuksia ovat näkymien päivittäminen, näkymien sulkeminen, tiedon esittäminen, virheilmoituksien esittäminen, etenemispalkin esittäminen ja mallin toteutus. MVVM-suunnittelumallin arvioinnissa keskitytään käyttöliittymäsovelluksen modulaarisuuden, suorituskyvyn, toteutettavuuden, ylläpidon ja opittavuuden parantamiseen.

Luvussa 2 käsitellään malli-näkymä-arkkitehtuureja yleisesti. Luvussa käydään läpi yhteisiä ominaisuuksia ja hyötyjä joita voidaan saavuttaa hyödyntämällä malli-näkymä-arkkitehtuureja. Luvussa esitellään kolme arkkitehtuuria MVC, MVP ja MVVM. Luvussa 3 perehdytään MVVM-suunnittelumallin toteuttamiseen WPF-teknologialla kirjallisuudessa esitettyjen teorioiden mukaisesti. Luvussa esitellään WPF-teknologian tärkeimmät ominaisuudet, jotka liittyvät MVVM-suunnittelumallin mukaisen sovelluksen toteuttamiseen. Lisäksi luvussa esitellään esimerkkisovellus, joka on toteutettu MVVM-suunnittelumallin mukaisesti. Luvussa 4 esitellään havaintoja, kuinka MVVM-suunnittelumallia voidaan soveltaa oikeassa ohjelmistoprojektissa. Luvussa esitellään toteutusratkaisuja, joita on tehty toteuttaessa MVVM-suunnittelumallin mukaista sovellusta ohjelmisto projektissa. Luvussa 5 arvioidaan, kuinka MVVM-suunnittelumallin soveltaminen projektissa onnistui. Suunnittelumallia arvioidaan modulaarisuuden, suorituskyvyn, toteutettavuuden, ylläpidon ja opittavuuden näkökulmista. Luvussa 6 esitellään yhteenveto tämän diplomityön tuloksista.

2 MALLI-NÄKYMÄ-ARKKITEHTUURIT

Ohjelmistosuunnittelijoille käyttöliittymäsovelluksien suunnittelu ja toteuttaminen asettaa erityisiä haasteita. Sovelluslogiikan lisäksi ohjelmistosuunnittelijoiden tulee toteuttaa tiedon esittäminen käyttäjälle, sekä toteuttaa vuorovaikutus käyttäjän ja järjestelmän välille. Käyttöliittymäsovelluksien oletetaan tarjoavan visuaalisesti miellyttäviä käyttäjäkokemuksia, ja samalla tehokasta ja intuitiivista tietojen käsittelyä ja hallintaa ilman turhauttavia varmennuksia tai latausaikoja. Järjestelmän tulisi automatisoida itsestään selvät toiminnot, mutta silti antaa käyttäjälle vaikutelman siitä, että käyttäjä on tietoinen järjestelmän tilasta ja pystyy hallitsemaan järjestelmän tapahtumia. Jo ensimmäisistä käyttöliittymäsovelluksista alkaen malli-näkymä-arkkitehtuurit ovat tarjonneet ohjelmistosuunnittelijoille hyväksi todettuja käytäntöjä ja toteutustapoja käyttöliittymäsovelluksien toteutukseen (Reenskaug 2007).

Tässä luvussa käsitellään malli-näkymä-arkkitehtuureihin liittyvää sanastoa ja termejä. Lisäksi perehdytään tunnetuimpiin malli-näkymä-arkkitehtuureihin, kuten MVC ja MVP, sekä havainnollistetaan, miksi malli-näkymä-arkkitehtuurit ovat merkittävässä asemassa käyttöliittymäsovelluksien kehityksessä. Luvun aikana huomataan, kuinka vanhoja malli-näkymä-arkkitehtuureja on vaikea toteuttaa tämän päivän ohjelmointiympäristöissä. Kappaleessa 2.3.4 esitellään uudenaikainen MVVM malli-näkymä-arkkitehtuuri. Jokaisen esitetyn malli-näkymä-arkkitehtuurin jälkeen kyseistä arkkitehtuurimallia arvioidaan toteutettavuuden, yleiskäyttöisyyden, testattavuuden, opittavuuden ja ylläpidettävyyden näkökulmista. Arvioinnissa käytetään kolmetasoista asteikkoa +, ++ ja +++. Kappaleessa 2.4 malli-näkymä-arkkitehtuurien arvioinnit on koottu yhdeksi taulukoksi.

Kirjallisuudessa luvataan (Anderson 2009; Smith 2009; Prism 2010; Garofalo 2011), että MVVM-suunnittelumalli mahdollistaa nykyaikaisten ohjelmointityökalujen ja käyttöliittymäkirjastojen hyödyntämisen sekä tehokkaan tavan toteuttaa visuaalisesti näyttäviä ja toiminnallisesti tehokkaita käyttöliittymäsovelluksia. Tämän työn seuraavissa luvuissa käsitellään MVVM-suunnittelumallin toteuttamista, soveltamista sekä syvällisempää arviointia sen todellisista hyödyistä.

2.1 Käsitteistö

2.1.1 Malli-näkymä-arkkitehtuuri

Tässä työssä malli-näkymä-arkkitehtuurilla tarkoitetaan yleistä käsitettä, joka kattaa kaikki arkkitehtuurit, joiden tarkoituksena on tarjota arkkitehtuuri- ja suunnitteluratkaisuja käyttöliittymäsovelluksien toteutukselle. Nimitys malli-näkymä syntyy siitä, että

arkkitehtuurien toteutuksen ydin perustuu näkymän erottamiseen sovelluslogiikasta, eli järjestelmän jakaminen malliin ja näkymään. Useasti näissä arkkitehtuureissa on myös ohjaimen tapainen osa, jonka vastuulla on näkymän ja mallin yhdistäminen. Kappaleessa 2.2 on käsitelty malli-näkymä-arkkitehtuurien tyypillisiä tehtäviä ja tavoitteita.

2.1.2 Malli

Mallilla tarkoitetaan tässä työssä järjestelmän osaa, jonka tehtävänä on tietomallin toteuttaminen, tiedon varastointi ja tiedon käsittelyyn liittyvien toimintojen toteuttaminen. Hyvin suunniteltu malli on monikäyttöinen ja kuvaa jotain tiettyä asiakokonaisuutta todellisesta maailmasta. Mallin tehtävänä on kuvata todellisen maailman käsitteitä ja tarjota toiminnallisuutta, joiden avulla eri käsitteiden suhteita ja vaikutuksia toisiinsa voidaan mallintaa.

2.1.3 Näkymä

Näkymä on järjestelmän osa, jonka käyttäjä voi havaita. Järjestelmä koostuu useista näkymistä. Järjestelmän kaikkien näkymien muodostamaa yhteistä kokonaisuutta kutsutaan käyttöliittymäksi. Näkymän tehtävänä on tarjota käyttäjälle työkalut, joiden avulla järjestelmää voidaan hallita. Järjestelmän hallintaan liittyy tiedon hakemista, esittämistä, sekä erilaisten toimintojen suorittamista.

2.1.4 Arkkitehtuurimalli

Arkkitehtuurimallilla tarkoitetaan tässä työssä teoreettista kuvausta järjestelmän osista ja niiden välisistä vuorovaikutuksista. Arkkitehtuurimallissa kuvataan yleisesti eri osat, niiden vastuut ja tehtävät. Arkkitehtuurimalli on yleiskäyttöinen ja tarjoaa korkean tason suunnitteluratkaisuja, johonkin tiettyyn ongelmaan. Arkkitehtuurimallin hyödyntäminen tarkoittaa, että arkkitehtuurimallissa kuvattuja asioita sovelletaan omaan ongelmaan. Arkkitehtuurimalli ei ota kantaa toteutusteknologiaan.

2.1.5 Arkkitehtuuri

Arkkitehtuuri on kuvaus toteutetun järjestelmän suunnitteluratkaisuista, komponenteista ja niiden välisistä vuorovaikutuksista sekä vastuista (Bass et al. 2003). Arkkitehtuurilla tarkoitetaan tässä työssä kuvausta toteutetusta arkkitehtuurimallista. Arkkitehtuuri on näin ollen tarkempi kuvaus siitä, kuinka jokin arkkitehtuurimalli on toteutettu jossain tietyssä järjestelmässä.

2.1.6 Suunnittelumalli

Suunnittelumalli (eng. design pattern) on matalan tason toteutusratkaisu. Suunnittelumalli kuvaa käytännössä, kuinka jokin ongelma ratkaistaan. Yleensä suunnittelumallit ovat tunnettuja ja käytännössä hyväksi havaittuja menetelmiä ja ratkaisuja, joilla tietty ongelma voidaan ratkaista tietyssä tilanteessa. Suunnittelumalli ei välttämättä ole sidok-

sissa johonkin tiettyyn teknologiaan, mutta on silti tarkka kuvaus toteutuksesta. (Gamma et al. 1994.)

2.2 Malli-näkymä-arkkitehtuurien tehtävät

Malli-näkymä-arkkitehtuurien tehtävänä on parantaa käyttöliittymäsovelluksien modulaarisuutta, toteutettavuutta, ylläpidettävyyttä sekä testattavuutta. Malli-näkymä-arkkitehtuurit tarjoavat ohjelmistosuunnittelijoille työkaluja, joiden avulla voidaan toteuttaa tehokkaita ja ylläpidettäviä käyttöliittymäsovelluksia.

Käyttöliittymäsovelluksen toteutuksessa on helppo päätyä ratkaisuun, jossa järjestelmän sovelluslogiikkaa toteutetaan jonkin tietyn näkymän yhteyteen. Ratkaisu on helppo, ja se on vaivaton toteuttaa. Samaa toiminnallisuutta ja tietosisältöä voi kuitenkin tarvita jokin toinen järjestelmän osa tai näkymä. Tästä seuraa, että viitteitä tietosisältöön on ympäri järjestelmää, ja lopulta mikään osa järjestelmästä ei ole täysin vastuussa tiedon hallinnasta. Kun tietosisällön rakenteeseen tai sen käsittelyyn tulee muutos, muutoksen aiheuttamat korjaukset pitää toteuttaa kaikkialle järjestelmään. Malli-näkymä-arkkitehtuurien yhteinen tehtävä on erottaa järjestelmän sovelluslogiikka sen käyttöliittymästä. Vastuualueiden jakaminen eri osille ja modulaarisuus on tärkeää, jotta tunnistetaan mihin osaan järjestelmästä tietyt toiminnallisuudet tai muutokset tulisi kohdistaa. Arkkitehtuurin merkitys kasvaa järjestelmän suuruuden kasvaessa. Pienissä järjestelmissä muutokset ja ylläpitotoimet on helppo hallita, mutta suurissa järjestelmissä muutosten hallinta on vaikeampaa ja muutosten toteuttaminen vie enemmän aikaa.

Järjestelmän testaaminen käyttöliittymän välityksellä vaatii usein testaajan toimenpiteitä. Mikäli sovelluslogiikkaa on toteutettu käyttöliittymän yhteyteen, on suuri osa järjestelmän testauksesta suoritettava käyttöliittymän välityksellä. Käyttöliittymätestauksen automatisointi on työlästä, koska käyttöliittymään kohdistuu usein visuaalisia muutoksia jolloin painikkeiden sijainnit muuttuvat. Muutoksien seurauksena myös testitapaukset tulee päivittää. Malli-näkymä-arkkitehtuurin ansiosta mallin toteuttama sovelluslogiikka on itsenäinen kokonaisuus, joka tarjoaa palveluita näkymille. Mallin tarjoamiin palveluihin voidaan toteuttaa automaattisesti ajettavia yksikkötestejä, jotka testaavat samat palvelut mitä näkymät käyttävät. Mallin rajapinta on staattisempi kuin järjestelmän käyttöliittymä, jolloin testitapauksia ei tarvitse päivittää niin useasti. Malli-näkymä-arkkitehtuurin mahdollistama testauksen automatisointi säästää aikaa ja resursseja.

Malli-näkymä-arkkitehtuurin ansiosta muutokset käyttöliittymään ja uusien näkymien tekeminen järjestelmään on helpompaa, koska käyttöliittymä ei sisällä sovelluslogiikkaa. Uusi näkymä voidaan toteuttaa esittämään tietomallista jotain uutta tietoa. Hyvin toteutetun arkkitehtuurin ansioista järjestelmään tehtävät korjaukset ja muutokset voidaan rajata selkeisiin kokonaisuuksiin, jolloin järjestelmän ylläpidettävyys paranee. (Buschmann et al. 1996; Potel 1996; Smith 2009; Garofalo 2011.)

2.3 Malli-näkymä-arkkitehtuurien esittely

2.3.1 Arviointi

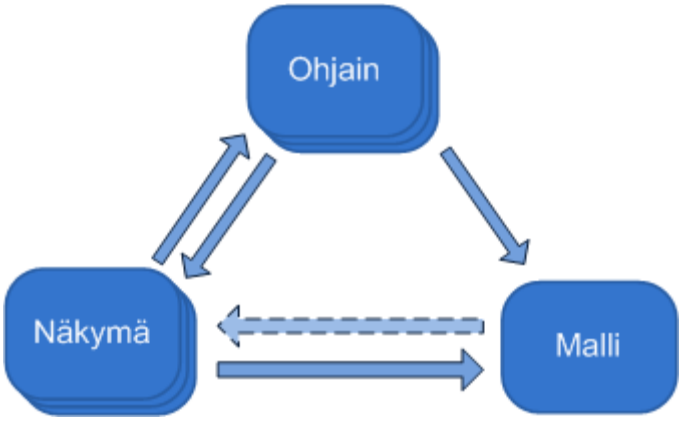
Kappaleessa 2.3 esitellään tunnetuimmat malli-näkymä-arkkitehtuurit. Kappaleessa esitettyjä malli-näkymä-arkkitehtuureja arvioidaan niiden ominaisuuksien perusteella. Seuraavassa listassa on esiteltynä arvioinnissa käytettävät ominaisuudet ja niiden kuvaukset.

- **Toteutettavuus:** Kuinka helppo arkkitehtuurimalli on käytännössä toteuttaa? Onko arkkitehtuurimallin toteuttamiseen työkaluja tai ohjeita? Vastaako teoreettinen kuvaus arkkitehtuurimallista käytännön todellisuutta?
- **Yleiskäyttöisyys:** Voidaanko arkkitehtuurimallia soveltaa eri ongelmiin? Voiko arkkitehtuurimallin toteuttaa eri teknologioilla?
- **Testattavuus:** Tarjoaako arkkitehtuurimalli lisäetua järjestelmän testaukselle? Mahdollistaako arkkitehtuurimalli automatisoidun testauksen?
- **Opittavuus:** Kuinka helposti arkkitehtuurimallin ydinajatuksen voi oppia? Kuinka helposti toteutettu arkkitehtuuri voidaan oppia ja ymmärtää? Onko kirjallisuudessa saatavilla tietoa arkkitehtuurimallin toteuttamisesta?
- **Ylläpidettävyys:** Kuinka helppo arkkitehtuurimallin mukaiseen järjestelmään on tehdä muutoksia? Mahdollistaako arkkitehtuurimalli uusien osien lisäämisen jälkikäteen?

Malli-näkymä-arkkitehtuurien arviointi toteutettiin arvioimalla jokaisen arkkitehtuurimallin ominaisuuksia asteikolla +, ++ ja +++. Arkkitehtuurimalleja arvioitiin kirjallisuudesta saatavan tiedon ja käytännön kokemuksien perusteella. Jokaisen arkkitehtuurimallin esittelyn jälkeen arkkitehtuuria arvioitiin itsenäisesti. Arvioinnin tulokset ja johtopäätökset on koottu yhteen kappaleessa 2.4.

2.3.2 MVC – Model-View-Controller

Malli-Näkymä-Ohjain (eng. MVC – Model View Controller) on ensimmäinen arkkitehtuurimalli, jonka tarkoitus on erottaa järjestelmän sovelluslogiikka käyttöliittymästä. Mallin on kehittänyt Trygve Reenskaug vuonna 1979 (Reenskaug 2011). MVC-arkkitehtuurimalli rakentuu kuvan 1 mukaisesti kolmesta osasta. Malli (*Model*) on esitys järjestelmän tietosisällöstä sekä sitä käsittelevästä sovelluslogiikasta. Näkymä (*View*) on esitystapa, jolla tietosisältö näytetään käyttäjälle. Ohjain (*Controller*) vastaa käyttäjän syötteiden hallinnasta. (Reenskaug 2007.)



Kuva 1. *MVC-arkkitehtuurimalli (Interactive Application Architecture Patterns 2007).*

MVC-arkkitehtuurimallissa malli on itsenäinen kokonaisuus, joka toteuttaa järjestelmän toiminnallisuuden tiedon käsittelylle ja varastoinnille. Mallin toteutuksessa ei oteta kantaa siihen, miten tieto esitetään käyttäjälle. Järjestelmään kuuluu aina yksi malli, jota näkymät ja ohjaimet hyödyntävät. Mallin toteutuksessa ei olla kiinnostuneita siitä, mitkä osat mallia käyttävät, vaan mallista tulisi toteuttaa järjestelmän yleiskäyttöinen tietosisältö. Tämä suunnitteluratkaisu mahdollistaa sovelluslogiikan erottamisen muusta järjestelmästä. (Buschmann et al. 1996; Reenskaug 2007.)

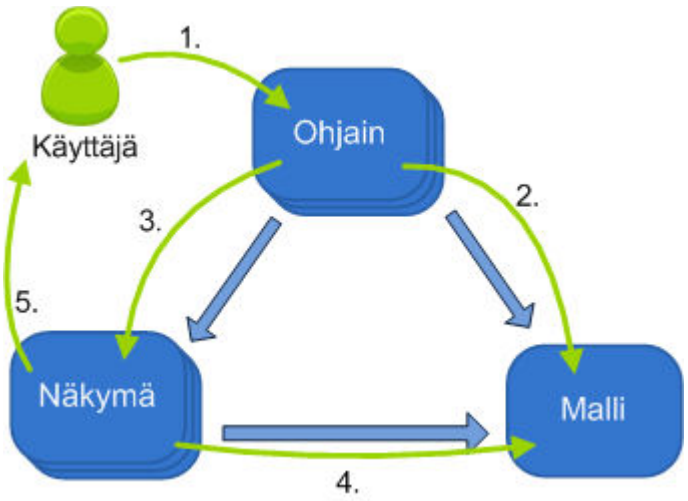
Malliin tulisi toteuttaa järjestelmän toiminnallisuutta, joka käsittelee järjestelmän tietoa ja laskentaa. Funktionaalinen toiminnallisuus, joka liittyy tiedon esittämiseen käyttäjälle, on ohjaimen tehtävä. Tästä seuraa, että sovelluslogiikka jakautuu kahteen eri kokonaisuuteen, tiedonkäsittelyyn ja tiedon esittämiseen. Tiedon käsittely on mallin tehtävä, ja tiedon esittäminen on ohjaimen ja näkymien tehtävä.

Näkymät ja ohjaimet muodostavat järjestelmän käyttöliittymän, ja toteuttavat tiedon esittämisen käyttäjälle. MVC-arkkitehtuurimallin mukaisesti näkymään kuuluu aina yksi ohjain, mutta ohjain voi ohjata useampaa näkymää. Ohjaimen tehtävänä on toimia välikappaleena käyttäjän ja järjestelmän välillä. Ohjain käsittelee kaikki järjestelmään syötteet, esimerkiksi hiiren ja näppäimistön painallukset. Ohjain tulkitsee käyttäjän syötteet, ja kutsuu näkymää ja mallia syötteitä vastaavilla toiminnallisuuksilla. Näkymän tehtävänä on esittää käyttäjälle järjestelmän tila. (Buschmann et al. 1996; Reenskaug 2007.)

Tiedonvälitys järjestelmän tietomallin ja käyttöliittymän välillä voidaan toteuttaa passiivisena tai aktiivisena. Passiivisessa toteutuksessa ohjaimen vastuulla on ilmoittaa näkymille, milloin järjestelmän tietosisältö on muuttunut, jolloin näkymät käyvät päivittämässä oman tietonsa mallista. Aktiivisessa toteutuksessa näkymät ovat sidottuina kuuntelemaan mallissa tapahtuvia muutoksia. (Garofalo 2011) Aktiivisen toteutuksen näkymien ja mallin välinen vuorovaikutus voidaan toteuttaa muun muassa tarkkailija-suunnittelumallin avulla (Fowler 2006). Tarkkailija-suunnittelumallin (eng. *Observer pattern*) mukaisesti näkymät voidaan asettaa mallin kuuntelijoiksi. Kun jonkin toiminnon seurauksena mallin tietosisältö muuttuu, se ilmoittaa muutoksesta näkymille. Nä-

kymät vastaanottavat tiedon mallin muutoksesta ja osaavat päivittää itsensä hakemalla uudet tiedot mallista. (Gamma et al. 1994, s. 293.)

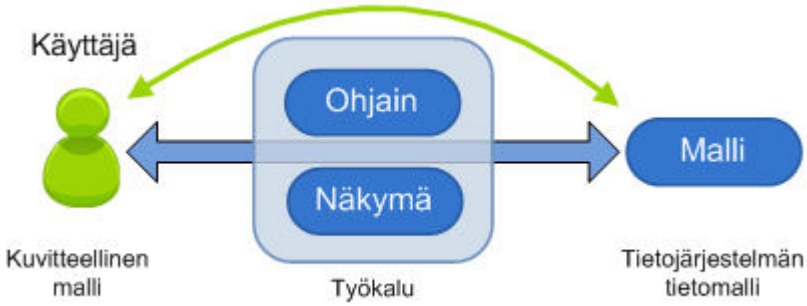
MVC-arkkitehtuurimallin mukainen käyttäjän syötteiden eteneminen on esitetty kuvassa 2.



Kuva 2. *Käyttäjän syötteiden eteneminen MVC-arkkitehtuurimallissa.*

1. Käyttäjä antaa järjestelmälle syötteen, jolla hän haluaa tietyn tuotteen tiedot.
2. Ohjain tulkitsee syötteen ja pyytää mallia hakemaan tuotteen tiedot.
3. Ohjain ilmoittaa näkymälle, että uudet tiedot voidaan päivittää.
4. Näkymä päivittää oman tietosisällön hakemalla tiedot mallista.
5. Käyttäjä saa syötteelleen vastauksen havainnoimalla näkymää.

Reenskaugin kuvaama MVC-arkkitehtuurimallin perimmäinen tarkoitus on luoda yhteys käyttäjien kuvitteellisen mallin ja tietojärjestelmän tietomallin välille. Mallien välillä toimii ohjaimesta ja näkymistä koostuva työkalu (eng. *Tool*). Työkalu edustaa jotain tiettyä näkemystä mallista ja antaa käyttäjälle mahdollisuuden hallita tietojärjestelmän tietomallia. Käyttäjä ei käytä tietojärjestelmän tietomallia suoraan vaan hyödyntää sen palveluita työkalun välityksellä, ks. kuva 3. Yhteen tietomalliin voi liittyä samanaikaisesti useita erilaisia työkaluja, eli käyttäjiä. Arkkitehtuurin ansiosta on mahdollista, että samanaikaisesti useampi käyttäjä saa vaikutuksen siitä, että käyttää samaa tietomallia suoraan vaikka todellisuudessa tietomallia käytetään erillisen työkalun välityksellä. Reenskaugin kuvaaman työkalun avulla voidaan toteuttaa järjestelmiä jotka hyödyntävät samaa tietomallia ja mahdollistavat tuen usealle käyttäjälle samanaikaisesti. (Reenskaug 2011.)



Kuva 3. *MVC-arkkitehtuurimallin olennainen tarkoitus (Reenskaug 2011).*

Reensgaugin MVC-arkkitehtuurimallilla on merkittävä asema malli-näkymä-arkkitehtuurien historiassa. Se on ensimmäinen arkkitehtuurimalli, joka tarjoaa ratkaisun näkymän ja sovelluslogiikan erottamiseen toisistaan. MVC-arkkitehtuurimalli on toiminut arkkitehtuurina ensimmäisien käyttöliittymien ja käyttöliittymäkirjastojen toteutuksessa Smalltalk-kielellä (Reenskaug 2011). MVC-arkkitehtuurimalli on kuvattu hyvin korkealla tasolla, joka tekee siitä yleiskäyttöisen. Yleiskäyttöinen kuvaus arkkitehtuurimallista antaa mahdollisuuden soveltaa ratkaisua moneen eri tilanteeseen. Tämä on aiheuttanut sen, että MVC-arkkitehtuurimallista on olemassa monta erilaista toteutusta. Osa väitetyistä MVC-arkkitehtuurimallin toteutuksista vastaa lopulta hyvin vähän alkuperäistä arkkitehtuurimallia.

Ohjelmointiympäristöt ja käyttöliittymien ohjelmointi on kehittynyt merkittävästi vuodesta 1979, jolloin MVC-arkkitehtuurimalli on julkaistu. Käyttäjän syötteiden hallinta ohjaimen avulla on suurin ongelma MVC-arkkitehtuurimallin mukaisen järjestelmän toteuttamisessa tänä päivänä. Ohjelmointiympäristöt sisältävät valmiita käyttöliittymäkomponentteja, jotka toteuttavat syötteiden vastaanottamisen ja hallinnan. Tämä tekninen ratkaisu aiheuttaa sen, että käyttäjän syötteiden hallinta on toteutettava näkymässä. Tästä seuraa, että MVC-arkkitehtuurimallin yksityiskohtainen toteuttaminen koko järjestelmän arkkitehtuurina tämän päivän ohjelmointiympäristöissä on mahdotonta.

MVC-arkkitehtuurimalli on tarjonnut ja tarjoaa edelleen hyvän teorian siitä, kuinka käyttöliittymäsovelluksien tarkoitus on luoda yhteys käyttäjien kuvitteellisen tietomallin ja sovelluksen tietojärjestelmän tietomallin välille. Tietomallien yhdistäminen on yhteinen tavoite kaikille malli-näkymä-arkkitehtuureille. MVC-arkkitehtuuri on toiminut pohjana monelle uudelle malli-näkymä-arkkitehtuurille. Yksi uudempi malli-näkymä-arkkitehtuuri on MVP, joka on kuvattu seuraavassa kappaleessa.

Seuraavassa listassa on arvioitu MVC-arkkitehtuurimallia

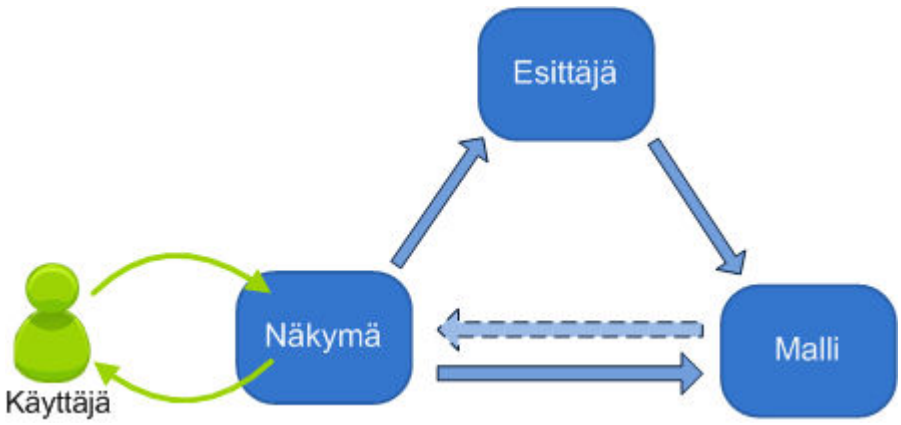
- **Toteutettavuus (+):** Alkuperäisen MVC-arkkitehtuurimallin toteuttaminen tämän päivän ohjelmointiympäristössä on mahdotonta, koska ohjelmointiympäristöjen käyttöliittymäkomponentit toteuttavat käyttäjän syötteiden hallinnan. MVC-arkkitehtuurimallia kuvaava kirjallisuus esittelee arkkitehtuurimallin kor-

kealla tasolla. Toteuttajalla on suuri vastuu toteutuksen onnistumisesta, koska arkkitehtuurin kuvaus voidaan ymmärtää monella tavalla.

- **Yleiskäyttöisyys (+++):** MVC-arkkitehtuurimalli on yleiskäyttöinen arkkitehtuuri. Arkkitehtuurimallin teoriaa voidaan hyödyntää monella eri tavalla moneen eri ongelmaan.
- **Testattavuus (++):** MVC-arkkitehtuurimalli jakaa järjestelmän kolmeen eri kokonaisuuteen. Jokaista eri osaa voidaan testata omina kokonaisuuksinaan. Arkkitehtuurimalli ei kuitenkaan tarjoa selkeää rajapintaa toiminnallisuuteen. Järjestelmän toiminnallisuuden muodostaa näkymän ja ohjaimen muodostamat parit.
- **Opittavuus (+):** MVC-arkkitehtuurimallin kuvaukset ovat korkealla tasolla, jolloin arkkitehtuurimallin voi ymmärtää monella eri tavalla. MVC-arkkitehtuurimallin mukaiset järjestelmät eivät välttämättä ole rakenteeltaan yhdenmukaisia.
- **Ylläpidettävyys (+):** Uusien näkymien lisääminen vaatii myös uusien ohjaimien toteuttamista.

2.3.3 MVP – Model-View-Presenter

Malli-Näkymä-Esittäjä (eng. MVP – Model View Presenter) on IBM:n kehittämä malli-näkymä-arkkitehtuuri. MVP tarjoaa yleiskäyttöisen arkkitehtuurimallin käyttöliittymäkomponenttien sekä käyttöliittymäsovelluksien kehitykseen. Ensimmäinen julkaisu on Mike Potelilta vuodelta 1996, jolloin MVP-arkkitehtuurimalli kehitettiin C++ ja Java ohjelmointikielille. MVP perustuu MVC-arkkitehtuurimalliin, ja siinä on paljon samoja piirteitä kuin vanhemmassa MVC-arkkitehtuurimallissa. Suurin ero on käyttäjän syötteiden käsittelyssä. MVP-arkkitehtuurimallissa näkymä vastaa käyttäjän syötteiden vastaanottamisesta. (Potel 1996.) MVP-arkkitehtuurimalli on esitetty kuvassa 4.



Kuva 4. *MVP-arkkitehtuurimalli (Potel 1996).*

MVP-arkkitehtuurimallissa näkymä (*View*) seuraa käyttäjän tekemiä toimintoja ja hallitsee niistä aiheutuvia tapahtumia. Näkymä ei toteuta logiikkaa toimintojen suorittamiseksi vaan kutsuu esittäjään (*Presenter*) toteutettuja toiminnallisuuksia. Esittäjä sisältää tarvittavan logiikan toiminnallisuuden suorittamiselle ja tekee tarvittavat muutokset malliin (*Model*). Malli tarjoaa toiminnallisuuden tiedon varastoinnille ja käsittelylle. Rakenteen ansiosta sovelluslogiikka ei ole sidoksissa näkymään. Esittäjään toteutettuja toiminnallisuuksia voidaan kutsua erilaisista käyttöliittymistä tai suoraan jonkin toisen ohjelman toimesta. Mallissa tapahtuvat muutokset välitetään tarkkailija-suunnittelumallin avulla näkymään. (Potel 1996.)

MVP-arkkitehtuurimallin tärkein periaate on jakaa järjestelmän vastuu käyttöliittymän ja tiedonhallinnan osa-alueisiin. Käyttöliittymän osa-alue on esittäjän ja näkymän vastuulla. Osa-alueen tehtävänä on vastata kysymykseen: Kuinka käyttäjä on vuorovaikutuksessa järjestelmän tiedon kanssa? Käyttöliittymän toteutuksessa on tärkeää tiedon esittäminen näkymässä ja käyttäjän syötteiden liittäminen sovelluslogiikan toimintoihin. Tiedonhallinnan osa-alue on esittäjän ja mallin vastuulla. Osa-alueen tehtävänä on vastata kysymykseen: Kuinka järjestelmän tietoa käsitellään? Tiedonhallinta koostuu käytettävissä olevan tiedon määrittämisestä, valitsemisesta sekä muutoksien tekemisestä. (Potel 1996.)

MVP-arkkitehtuurimallin toteutus nykyisillä ohjelmointikielillä ja -ympäristöillä on suoraviivaisempaa kuin MVC-arkkitehtuurimallin toteutus. MVP-arkkitehtuurimalli tukee kehitystä, jossa näkymän käyttöliittymäkomponentit toteuttavat käyttäjän syötteiden hallinnan. Toinen merkittävä etu MVP-arkkitehtuurimallissa on esittäjän toteuttama rajapinta, joka kapseloi järjestelmän sovelluslogiikan. Esittäjän avulla sovelluslogiikka ja tiedonhallinta on rajattu selkeäksi kokonaisuudeksi. Esittäjän tarjoamaa toiminnallisuutta on helppo hyödyntää uusista näkymistä tai kokonaan eri järjestelmien osista. MVP-arkkitehtuurimallin mukaisen järjestelmän testaus voidaan automatisoida kirjoittamalla testitapauksia, jotka testaavat esittäjän toiminnallisuuksia. Suuri osa järjestelmätestauksesta voidaan kattaa automaattisilla testitapauksilla, koska näkymät ja muut järjestelmän osat hyödyntävät näitä samoja toiminnallisuuksia. (Potel 1996; Fowler 2006.)

Seuraavassa listassa on arvioitu MVP-arkkitehtuurimallia

- **Toteutettavuus (++):** MVP-arkkitehtuurimalli voidaan toteuttaa tämän päivän ohjelmointiympäristöillä. Arkkitehtuurimallin kuvaukset ovat kuitenkin korkealla tasolle esitettyjä, joten toteuttajan tulee ratkaista tekniset yksityiskohdat.
- **Yleiskäyttöisyys (++):** MVP-arkkitehtuurimallin yleiskäyttöisyys on rajoittuneempi kuin MVC-arkkitehtuurimallin.
- **Testattavuus (+++):** MVP-arkkitehtuurimallin esittäjä toteuttaa rajapinnan, jonka avulla järjestelmän sovelluslogiikkaa voidaan testata.
- **Opittavuus (+):** MVP-arkkitehtuurimallin kuvaukset ovat korkealla tasolla, jolloin arkkitehtuurimallin voi ymmärtää monella eri tavalla. MVP-arkkitehtuurimallin mukaiset järjestelmät eivät välttämättä ole rakenteeltaan yhdenmukaisia.

- **Ylläpidettävyys (++):** MVP-arkkitehtuurimallin esittäjä kapseloi järjestelmän sovelluslogiikan, joten järjestelmään tehtävien muutoksien seuraukset voidaan rajata tiettyyn osaan järjestelmästä.

2.3.4 MVVM – Model-View-ViewModel

Malli-Näkymä-Näkymämalli (eng. MVVM – Model View ViewModel) on tässä luvussa käsiteltävistä malli-näkymä-arkkitehtuureista uusin. MVVM on eniten teknologia riippuvainen arkkitehtuuri ja eroaa näin ollen edellä esitetyistä muista korkean tason malli-näkymä-arkkitehtuureista. MVVM ottaa kantaa niin matalan tason toteutusratkaisuihin, kuin myös korkean tason arkkitehtuuri rakenteeseen. Kirjallisuudessa, ja myös tässä työssä, MVVM malli-näkymä-arkkitehtuuria kutsutaan MVVM-suunnittelumalliksi (eng. *MVVM design pattern*). (Smith 2009; Garofalo 2011.)

MVVM-suunnittelumallin on kehittänyt Microsoft, ja se on peräisin MVP-arkkitehtuurimallista. Microsoftin uusi WPF-käyttöliittymäkirjasto on suunniteltu ja toteutettu niin, että WPF-teknologian kaikki ominaisuudet saadaan tehokkaasti hyödynnettyä käyttämällä MVVM-suunnittelumallia. Suunnittelumallia voidaan soveltaa myös muihin toteutusteknologioihin, mutta WPF-teknoligian tapa määritellä käyttöliittymän XAML-kuvauskielellä on avainasemassa toteutettaessa MVVM-suunnittelumallin mukaista järjestelmää. MVVM-suunnittelumallin toteutusta ja soveltamista jatketaan tämän työn seuraavissa luvuissa. Tässä luvussa keskitytään suunnittelumallin yleiseen rakenteeseen. (Smith 2009; Garofalo 2011.)

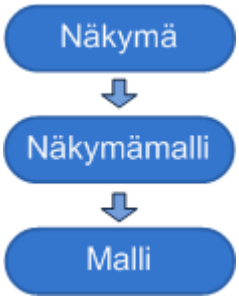
MVVM-suunnittelumalli muodostuu kuvan 5 mukaisesti näkymästä (*View*), näkymämallista (*ViewModel*) sekä mallista (*Model*).

Kuva 5: *MVVM-suunnittelumalli (Prism 2010, s. 53).*

Malli kuvaa järjestelmän tietomallia. Mallin tehtävänä on varastoida järjestelmän tieto ja mahdollistaa sen käyttö. Näkymä muodostaa järjestelmän käyttöliittymän ja tarjoaa käyttäjälle mahdollisuuden havainnoida järjestelmän tietoa ja suorittaa toimenpiteitä. Näkymämalli yhdistää järjestelmän mallin ja näkymän. Sen tehtävänä on toteuttaa käyttäjälle tarjotut toiminnallisuudet ja huolehtia malliin kohdistuvista muutoksista. Näkymämalli kapseloi järjestelmän sovelluslogiikan ja tarjoaa näkymille rajapinnan toimintojen suorittamiselle ja tiedon esittämiselle. (Prism 2010; Garofalo 2011, ss. 39-43.)

MVVM hyödyntää Fowlerin (2004) esittelemää PM-suunnittelumallia (eng. *PM – Presentation Model*). PM-suunnittelumallin ydin on erottaa näkymien toteutus visuaaliseen toteutukseen sekä toiminnalliseen käyttäytymiseen (Fowler 2004). MVVM suunnittelumallissa näkymä toteuttaa käyttöliittymien visuaalisen ulkonäön. Näkymän tehtävänä on määritellä, minkälaisista komponenteista näkymä muodostuu ja mitä toimintoja näkymässä on mahdollista suorittaa. Näkymämallin tehtävänä on tarjota rajapinta järjestelmän tietomalliin ja toiminnallisuuteen, joihin näkymä tai näkymät voivat sitoutua. MVVM-suunnittelumallissa näkymämallit hallitsevat myös näkymien tilatietoa ja tarjoavat näkymille tiedon, milloin tietyt toiminnot ovat käytettävissä. (Smith 2009; Garofalo 2011).

MVVM-suunnittelumallin osat muodostavat kuvan 6 mukaisen kerrosarkkitehtuurin.



Kuva 6: *Osien väliset suhteet MVVM-suunnittelumallissa (Anderson 2009).*

MVVM-suunnittelumalli perustuu siihen, että eri osat tunnistavat vain kuvassa 6 alempana olevia osia. Malli tarjoaa yleiskäyttöisen kuvauksen järjestelmän tietomallista, mutta ei tiedä, mitkä osat tai järjestelmät sitä käyttävät. Näkymämalli kapseloi mallin ja toteuttaa lisää tarvittavaa toiminnallisuutta ja sovelluslogiikkaa. Näkymämalli tuntee mallin, jonka avulla toteuttaa toiminnallisuutta, mutta näkymämalli ei tiedä, kuka sen toteuttamia palveluita hyödyntää. Kerrosarkkitehtuurijaon ansiosta MVVM-suunnittelumallin mukaiselle järjestelmälle voidaan toteuttaa uusia näkymiä helposti, koska näkymät hyödyntävät vain näkymämallien tarjoamia palveluita. (Anderson 2009, ss. 373-402.)

MVVM-suunnittelumallin muodostama modulaarisuus on merkittävä hyöty, joka saavutetaan hyödyntämällä suunnittelumallia. Selkeän rakenteen ansiosta järjestelmästä saadaan ylläpidettävä ja muutoksien tekeminen helpottuu. MVVM-suunnittelumallin mukaisesti näkymä on näkymämallin tarkkailija, joka mahdollistaa uusien näkymien toteuttamisen ja vanhojen muokkaamisen ilman muutoksia sovelluslogiikkaan tai tietomalliin. Suunnittelumalli mahdollistaa myös näkymämallien uudelleenkäytettävyyden. (Anderson 2009; Smith 2009; Garofalo 2011.)

WPF-teknologia ja XAML-kuvauskieli mahdollistavat tehokkaan ja helppokäyttöisen tavan toteuttaa MVVM-suunnittelumallin mukaisen järjestelmän. WPF-teknologian tarjoamat uudet ominaisuudet tukevat MVVM-suunnittelumallin periaatteita. WPF-

teknologian avulla MVVM-suunnittelumalli on helposti toteutettavissa. WPF-teknologian ansiosta MVVM-suunnittelumallin edut ovat mahdollista saavuttaa myös käytännössä. (Anderson 2009; Smith 2009; Garofalo 2011.)

Malli-näkymä-arkkitehtuurien ongelma on niiden tulkitseminen. Jokainen toteuttaja tulkitsee arkkitehtuurit omalla tavallaan, johtuen osaksi siitä, että arkkitehtuureja on vaikea dokumentoida. Arkkitehtuurit antavat yleensä vain teoreettisen kuvan järjestelmän rakenteesta, jolloin käytännön toteutusratkaisut ovat täysin toteuttajan vastuulla. Tärkeää on, että arkkitehtuurien perimmäinen tarkoitus säilytetään toteutuksessa. MVC ja MVP malli-näkymä-arkkitehtuurit on kuvattu korkealla tasolla, ja ne eivät ota kantaa yksityiskohtaisiin toteutusratkaisuihin. MVVM-suunnittelumalli poikkeaa näistä malli-näkymä-arkkitehtuureista. MVVM-suunnittelumalli on kehitetty toteutettavaksi WPF-teknologialla, ja se tarjoaa myös matalan tason toteutusratkaisuja oikeaoppisen MVVM-suunnittelumallin toteutukselle. (Smith 2009; Prism 2010.)

Seuraavassa listassa on arvioitu MVVM-suunnittelumallia

- **Toteutettavuus (+++):** MVVM-suunnittelumallin mukaisen järjestelmän toteuttamiseen on saatavilla tarkat ohjeet. WPF-teknologia toteuttaa monia teknisiä työkaluja, joiden avulla MVVM-suunnittelumallin hyödyt voidaan käytännössä saavuttaa.
- **Yleiskäyttöisyys (++):** MVVM-suunnittelumalli on tarkoitettu toteutettavaksi WPF-teknologialla. Suunnittelumallin toteuttaminen jollakin muulla ohjelmointikielellä lisää toteutettavien yksityiskohtien määrää.
- **Testattavuus (+++):** MVVM-suunnittelumalli jakaa järjestelmän selkeisiin kokonaisuuksiin joita voidaan testata. Näkymämallit toteuttavat järjestelmän sovelluslogiikan ja näkymien tilatietoa, joten järjestelmän toiminnallisuuteen voidaan toteuttaa automaattisia testitapauksia.
- **Opittavuus (++):** MVVM-suunnittelumallin mukaisen järjestelmä toteuttamiseen löytyy hyvin ohjeita. MVVM-suunnittelumallin mukaiset järjestelmät toteuttavat saman rakenteen, jolloin toteutetun järjestelmän rakenne on helppo oppia ja ymmärtää.
- **Ylläpidettävyys (+++):** MVVM-suunnittelumallissa näkymät ovat tapa esittää tietoa näkymämalleista, joten uusien näkymien luominen järjestelmään on helppoa. Näkymämalleja voidaan uudelleenkäyttää esittämällä samaa tietoa järjestelmän toisessa näkymässä.

2.4 Malli-näkymä-arkkitehtuurien arviointi

Taulukon 1 perusteella MVVM-suunnittelumalli on soveltuvin vaihtoehto malli-näkymä-arkkitehtuuriksi. MVP ja MVVM ovat rakenteeltaan samantapaisia, mutta MVVM-suunnittelumallin toteutettavuus ja ylläpidettävyys ovat parempia. MVVM-suunnittelumalli kokoaa yhteen vanhojen malli-näkymä-arkkitehtuurien hyviä puolia ja tarjoaa hyvät työkalut arkkitehtuurimallin toteuttamiseen. MVVM-suunnittelumallin hyvä toteutettavuus mahdollistaa malli-näkymä-arkkitehtuurista saatavien hyötyjen toteutumisen myös käytännössä.

Taulukko 1. *Malli-näkymä-arkkitehtuurien ominaisuuksien arviointi.*

	MVC	**MVP**	**MVVM**
Toteutettavuus	+	++	+++
Yleiskäyttöisyys	+++	++	++
Testattavuus	++	+++	+++
Opittavuus	+	+	++
Ylläpidettävyys	+	++	+++
	8	10	13

3 MVVM-SUUNNITTELUMALLIN ESIMERKKI-TOTEUTUS

Tässä luvussa käydään läpi, kuinka MVVM-suunnittelumallin mukainen esimerkkisovellus toteutetaan WPF-teknologialla. Tavoitteena on antaa lukijalle tarvittavat lähtötiedot WPF-teknologian tarjoamista ominaisuuksista MVVM-suunnittelumallin mukaisen järjestelmän toteuttamiselle. Kappaleessa 3.1 esitellään esimerkkisovelluksen ominaisuudet. Seuraavassa kappaleessa 3.2 käydään läpi WPF-teknologian ominaisuudet, jotka liittyvät MVVM-suunnittelumallin toteuttamiseen. Viimeisessä kappaleessa 3.3 kuvataan, kuinka esimerkkisovellus on toteutettu MVVM-suunnittelumallin mukaisesti.

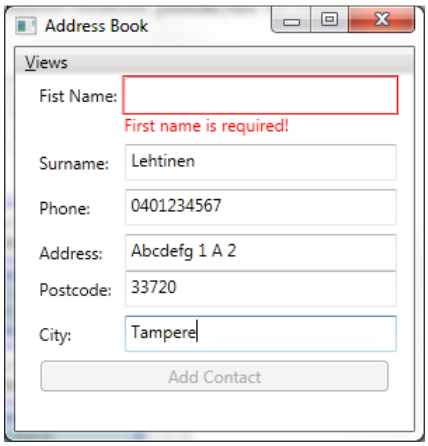
3.1 Puhelinluettelosovellus

Puhelinluettelosovellus on esimerkkisovellus, jonka avulla havainnollistetaan MVVM-suunnittelumallin toteutusta WPF-teknologialla. Sovelluksen suunnitteluratkaisut ja toteutusyksityiskohdat ovat peräisin kirjallisuudessa kuvatuista tavoista toteuttaa MVVM-suunnittelumallin mukainen sovellus. Tavoitteena on kuvata yksinkertainen MVVM-suunnittelumallin mukainen esimerkkisovellus ja antaa lukijalle lähtötiedot, kuinka yksinkertainen MVVM-suunnittelumallin mukainen sovellus toteutetaan.

Puhelinluettelosovellus koostuu kahdesta näkymästä. Päänäkymä listaa tiedot sovellukseen lisätyistä yhteyshenkilöistä. Yhteyshenkilöitä voidaan hakea näkymässä annettavan avainsanan avulla. Puhelinluettelosovelluksen päänäkymä on esitetty kuvassa 7.

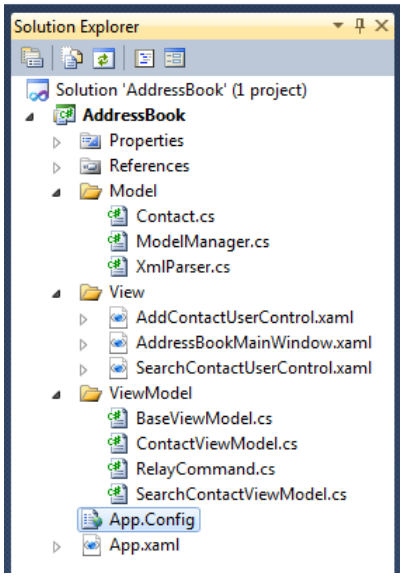
Kuva 7. *Puhelinluettelosovelluksen päänäkymä.*

Toinen näkymä mahdollistaa yhteyshenkilöiden lisäämisen sovellukseen. Näkymä on esitetty kuvassa 8. Yhteyshenkilöistä voidaan tallentaa etunimi, sukunimi, puhelinnumero ja osoite.



Kuva 8. *Näkymä yhteyshenkilön lisäämiseen puhelinluettelosovelluksessa.*

Puhelinluettelosovelluksen malli koostuu luokasta, joka määrittelee yhteyshenkilön tiedot. Puhelinluettelosovellukseen liittyvät tiedostot on listattu kuvassa 9.



Kuva 9: *Puhelinluettelosovelluksen tiedostot.*

3.2 Windows Presentation Foundation

Windows Presentation Foundation, eli WPF on Microsoftin tarjoama uusi käyttöliittymäkirjasto, joka korvaa vanhan *Windows Forms*-käyttöliittymäkirjaston. Tässä kappaleessa käydään läpi WPF-teknologian tärkeimmät ominaisuudet, jotka liittyvät MVVM-suunnittelumallin toteutukseen. WPF-teknologia pitää sisällään paljon muitakin ominaisuuksia kuin tässä kappaleessa esitetyt asiat. Lisätietoja WPF-teknologiasta voi lukea seuraavista lähteistä: MacDonald, 2010; Windows Presentation Foundation, 2011.

3.2.1 XAML

WPF-teknologia tarjoaa uuden tavan määritellä sovelluksien käyttöliittymiä. WPF-teknologia hyödyntää käyttöliittymien määrittämiseen XAML-kuvauskieltä (eng. *Extensible Application Markup Language*). XAML on XML pohjainen kuvauskieli, jolla muodostetaan puurakenne .Net olioista. XAML-käyttöliittymäkuvauksia voidaan toteuttaa graafisilla suunnittelutyökaluilla. Muun muassa Visual Studio tarjoaa graafisen suunnittelutyökalun käyttöliittymien toteutukseen. XAML-kuvauskielellä toteutettuja käyttöliittymiä on mahdollista toteuttaa myös kirjoittamalla käsin XAML-koodia. XAML-kuvauskieli ei ole ainoastaan WPF-teknologian hyödyntämä menetelmä. XAML-kuvauskielestä on olemassa useita muunnelmia, joita hyödynnetään erilaisissa käyttötarkoituksissa kuten sähköisissä dokumenteissa (XPS XAML) ja *Windows Workflow Foundation* elementtien määrittämiseen (WF XAML). (MacDonald 2010, ss. 23-60; XAML Overview 2011.)

WPF-teknologiassa XAML-kuvauskielen elementti vastaa aina jotain määriteltyä .Net-luokkaa. Kuvauskielen avulla .Net-luokista voidaan rakentaa puurakenne, joka kuvaa luokkien rakennetta toisiinsa. Kuvauskielien avulla voidaan rakentaa näkymiä, jotka koostuvat käyttöliittymäkomponenteista joiden sisällä on uusia käyttöliittymäkomponentteja. Luokkamäärittelyn lisäksi kuvauskielellä voidaan määritellä luokan ominaisuuksia. (MacDonald 2010, ss. 23-60.) Yksinkertainen WPF XAML-dokumentti on kuvattu esimerkissä 1.

```
<Window x:Class="Examples.XAMLExample"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
       Title="XAMLExample" Height="300" Width="300">
    <StackPanel>
        <TextBox
            Background="Gray" FontFamily="Courier New" FontSize="14"
            HorizontalAlignment="Right" IsReadOnly="True"
            Name="textBoxExample" Text="XAML example" />
    </StackPanel>
</Window>
```

Esimerkki 1. *Ikkunan määritys XAML-kuvauskielellä.*

Esimerkistä voidaan nähdä, kuinka dokumentissa on määritelty yksi ikkuna (*Window*). Kaikissa XAML-dokumenteissa on määritelty nimiavaruudet *xmlns* ja *xmlns:x*, jotka määrittelevät WPF-komponenttien sijainnin sekä XAML ominaisuuksia joilla voidaan vaikutta kuvausmäärittelyn tulkintaan. Suurin osa .Net luokkien ominaisuuksista voidaan asettaa kuvauskielessä tekstimuotoisella merkkijonolla. Esimerkissä 1 *TextBox*-luokalle asetetaan muun muassa seuraavia ominaisuuksia *Background*, *FontFamily*, *FontSize*, *HorizontalAlignment* ja *IsReadOnly*. XAML kuvauksessa kaikkien ominaisuuksien arvot annetaan merkkijonolla. Todellisuudessa kaikki ominaisuuksille annetut parametrit edustavat eri tyyppejä. *Background* on *Brush*-olio. *FontFamily* on merkkijono, joka annetaan *FontFamily*-luokan rakentajalle. *FontSize* on integer-muuttuja. *HorizontalAligment* on enum-tietotyyppi. *IsReadOnly* on boolean-muuttuja. XAML hyödyntää ominaisuuksien asettamiseen .Net-kirjaston tyyppi muuntimia (eng. Type Converters). Automaattisen tyyppimuunnoksen ansioista XAML-kuvauskielellä voidaan pienellä määrällä tekstiä kuvata monimutkaisiakin rakenteita. (MacDonald 2010, ss. 23-60.) Esimerkissä 2 on kuvattu saman ikkunan määritys käyttäen C#-koodia.

```
Window window = new Window();
window.Title = "XAMLExample";
window.Height = 300;
window.Width = 300;

StackPanel panel = new StackPanel();

TextBox textBox = new TextBox();
textBox.Background = new SolidColorBrush(Colors.Gray);
textBox.FontFamily = new FontFamily("Courier New");
textBox.FontSize = 14;
textBox.HorizontalAlignment = System.Windows.HorizontalAlignment.Right;
textBox.IsReadOnly = true;
textBox.Name = "textBoxExample";
textBox.Text = "XAML example";

panel.Children.Add(textBox);
window.Content = panel;
window.Show();
```

Esimerkki 2. *Ikkunan määritys C#-koodilla.*

XAML-kuvauskielen tehokkuus korostuu määriteltäessä monimutkaisempia ominaisuuksia, kuten listoja ja tiedon sitomista. Listojen esittelemiseen sekä tiedon sitomiseen palataan myöhemmin tässä työssä (ks. kappale 3.3 Puhelinluettelosovelluksen toteutus MVVM-suunnittelumallilla).

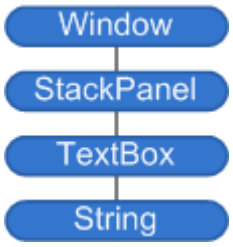
XAML tiedostojen liittäminen ajettavaan ohjelmaan tapahtuu kaksivaiheisen käännöksen tuloksena. Ensimmäisessä vaiheessa XAML tiedostosta käännetään binäärinen XAML tiedosto BAML. Käännöksen yhteydessä syntyy myös g.cs-päätteinen tiedosto, joka sisältää toteutuksen, jossa ladataan käännetty BAML-tiedosto, esitellään käytetyt komponentit ja kytketään tapahtumankäsittelijät. Käännöksen toisen vaiheen suorittaa ohjelmointikielen kääntäjä, jonka tuloksena on ajettava exe-tiedosto. Käännetyt BAML-tiedostot ovat staattisia resursseja käännetyssä binäärissä. XAML-tiedostoja voidaan ladata sovellukseen myös dynaamisesti ilman kääntämistä. XAML-tiedoston lataaminen dynaamisesti on kuitenkin hitaampaa kuin käännetyn BAML-tiedoston lataaminen. XAML-tiedostojen kääntämisestä voi lukea enemmän lähteestä MacDonald (2010, s. 48-56).

XAML-kuvauskielen hyödyntäminen näkymien määrittämisessä mahdollistaa sen, että näkymien toteutus ei ole sidoksissa sovelluksen sovelluslogiikan toteutukseen. XAML-kuvauskielen tarkoitus on määritellä sovelluksen käyttöliittymä niin, että se on helposti päivitettävissä ja tarvittaessa vaihdettavissa kokonaan ilman muutoksia sovelluksen sovelluslogiikkaan. XAML-kuvauskielellä toteutettu näkymä ottaa kantaa vain siihen, kuinka sovelluksen tieto esitetään näkymässä ja mitä toiminnallisuuksia käyttäjälle tarjotaan näkymästä tietomalliin. WPF-teknologian mukaan näkymä ei ole paikka, jossa sovelluksen tietoa säilytetään tai missä toteutetaan sovelluksen toiminnallisuutta.

Avainasemassa näkymien ja sovelluslogiikan liittämisessä toisiinsa on tiedon sitominen (ks. kappale 3.2.4 Tiedon sitominen) ja tietomallin asettaminen. Tietomalli asetetaan näkymälle käyttämällä *DataContext*-määrettä. *DataContext* kuvaa näkymän tietomallia. Tietomalli toteuttaa kaikki ominaisuudet ja toiminnallisuudet, joita XAML-kuvauskiellä toteutetussa näkymässä käytetään. Näkymissä määritellään vain ominaisuuksien nimiä, joiden sisältöä halutaan näyttää näkymissä. WPF-teknoligian mekanismien avulla sovelluslogiikan toteutus voidaan kapseloida erilliseen luokkaan, jota näkymät hyödyntävät. MVVM-suunnittelumallissa tätä luokkaa kutsutaan näkymämalliksi.

3.2.2 Looginen ja visuaalinen puu

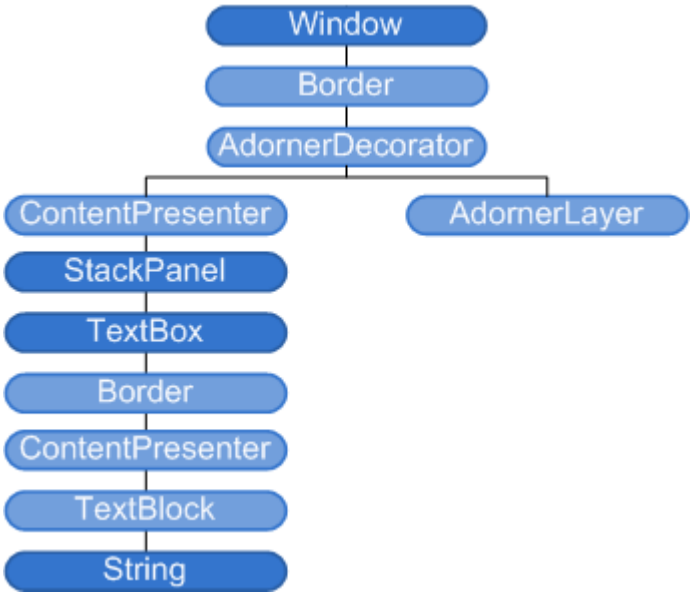
XAML-käyttöliittymäkuvauksien yhteydessä puhutaan usein sen puurakenteesta. XAML käyttöliittymä voidaan kuvata loogisena sekä visualisena puuna. Looginen puurakenne muodostuu käyttöliittymässä käytetyistä käyttöliittymäkomponenteista ja niiden rakenteesta. Sivulla 20 esitetyn esimerkin 1 muodostama looginen puu on kuvan 10 mukainen. (MacDonald 2010, ss. 119-158, ss. 499-543).



Kuva 10: *Looginen puurakenne.*

Looginen puurakenne ei vaikuta sovelluksen toteutukseen. XAML-kuvauskielellä toteutettu käyttöliittymä muodostaa automaattisesti loogisen puurakenteen. Loogista puurakennetta hyödynnetään tapahtumien välittämiseen. WPF-teknologian tapahtumien välitystä on esitelty kappaleessa 3.2.3 Reitietyt tapahtumat. (MacDonald 2010, ss. 119-158, ss. 499-543).

Visuaalinen puu tarjoaa mahdollisuuden vaikuttaa käyttöliittymäkomponenttien esitystapaan. Esimerkin 1 muodostama visuaalinen puu on esitelty kuvassa 11.



Kuva 11: *Visuaalinen puurakenne.*

Kuvasta 11 nähdään, että visuaalinen puu tarjoaa muun muassa luokkia, joiden avulla komponentin ulkoasua voidaan muuttaa. Komponenteille voidaan asettaa tyyliresursseja, jotka kuvaavat kuinka komponentti piirretään näytölle. Staattiset tyyliresurssit mahdollistavat yhdenmukaisen ulkoasun koko sovelluksessa. Komponenteille voidaan asettaa myös ehtoja, jotka toteutuessaan asettavat komponentille tietyn tyylin. (MacDonald 2010, ss. 119-158, ss. 499-543). XAML-kuvauskielen visuaalisen puun muokkaamisesta on esimerkkejä myöhemmin tässä työssä (ks. luku 3.3.2 AddressBookMainWindow-näkymä ja luku 3.3.3 AddContactUserControl-näkymä).

3.2.3 Reititetyt tapahtumat

Reititetyt tapahtumat (eng. *Routed Events*) ovat WPF-teknologian uudenlainen tapa käsitellä tapahtumia. Reititetyt tapahtumat tarjoavat tehokkaan tavan määritellä tapahtumia näkymissä. Reititettyjä tapahtumia hyödynnetään muun muassa tiedon sidonnassa (ks. kappale 3.2.4 Tiedon sitominen) sekä komentojen välittämisessä näkymästä sovelluslogiikalle (ks. kappale 3.2.7 Toiminnon suorittaminen). WPF-teknologian tapahtumat voidaan jakaa kolmeen kokonaisuuteen. (MacDonald 2010, ss. 119-158).

- **Suorat tapahtumat (Direct events):** Perinteiset .Net ohjelmoinnissa käytetyt tapahtumat ovat suoria tapahtumia. Tapahtuma laukaistaan määritetyssä luokassa ja toiset luokat voivat sitoutua kuuntelemaan tapahtumaa toteuttamalla tapahtumankäsittelijän. Tapahtuman laukaisu ei etene muille luokille. (MacDonald 2010, ss. 119-158).

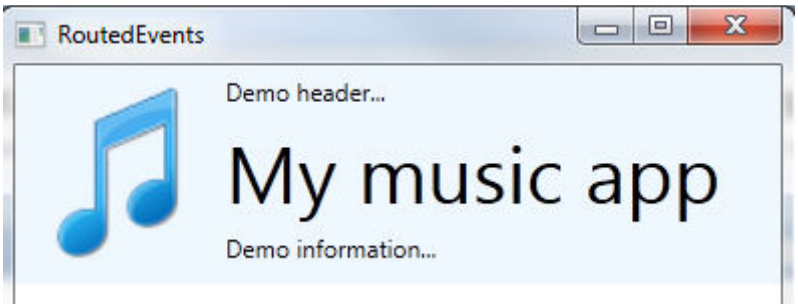
- **Bubbling-tapahtumat (Bubbling events):** Tapahtuman laukaisu aiheuttaa tapahtumapyyntöketjun, joka etenee loogista puurakennetta ylöspäin. Tapahtumapyyntöketjun aikana kaikki puurakenteessa määritetyt tapahtumankäsittelijät suoritetaan. Esimerkiksi *MouseUp*-tapahtuma on *bubbling*-tapahtuma. *Bubbling*-tapahtumaa havainnollistetaan seuraavassa esimerkissä 3. (MacDonald 2010, ss. 119-158).
- **Tunneling-tapahtumat (Tunneling events):** Tapahtuma vaeltaa loogista puurakennetta alaspäin kunnes saavuttavat oman määränpäänsä. Matkan aikana tapahtumalle voidaan asettaa valmistelevia toteutuksia. Esimerkiksi *PreviewKeyDown*-tapahtuma on *tunneling*-tapahtuma. Tapahtuman eteneminen alkaa korkeimmalta tasolta loogista puuta eli ikkunasta. Tapahtuma etenee loogista puurakennetta edelleen seuraaville komponenteille kunnes saavuttaa päämääränsä. *PreviewKeyDown*-tapahtuman päämäärä on se komponentti jossa käyttöliittymän kohdistus oli kun painiketta painettiin. *Tunneling*-tapahtuman ansiosta näppäimen painallus voidaan käsitellä ylemmällä tasolla tai se voidaan estää kokonaan. (MacDonald 2010, ss. 119-158).

Reititetyt tapahtumat ovat sidoksissa WPF-teknologian tapaa määritellä näkymiä XAML-kuvauskielellä. Esimerkissä 3 on määritelty alue, joka määrittää sovelluksen otsikon.

```
<Grid>
   ...
   <Label ... Background="AliceBlue" MouseUp="Label_MouseUp" >
      <Grid>
         ...
         <Image ... Source="/Examples;component/image.png"/>
         <Label ... Content="Demo header..." Padding="0" />
         <Label ... Content="My music app" FontSize="40" Padding="0" />
         <Label ... Content="Demo information..." Padding="0" />
      </Grid>
   </Label>
</Grid>
```

Esimerkki 3. *Reititetyt tapahtumat.*

Otsikko koostuu *Label*-komponentista, jonka sisälle on määritelty yksi *Image*-komponentti kuvan esittämistä varten ja kolme *Label*-komponenttia otsikon tekstikuvauksia varten. XAML-kuvauskielen muodostama käyttöliittymä on esitetty kuvassa 12.



Kuva 12. *Esimerkin 3 muodostama käyttöliittymä.*

Otsikkoaluetta painettaessa halutaan toteuttaa jotain tiettyä toiminnallisuutta. Perinteisesti .Net ohjelmoinnissa jokaiselle otsikkoalueen komponentille pitäisi määritellä *MouseUp*-tapahtuma ja sitoa se samaan tapahtumankäsittelijään. WPF-teknologiassa reititettyjen tapahtumien ansiosta riittää, että *MouseUp*-tapahtuma on määritelty ainoastaan ylimmässä *Label*-komponentissa. Mikäli käyttäjä painaa jotain tämän komponentin sisällä olevista komponenteista, *MouseUp*-tapahtuma laukeaa ja kulkeutuu *bubbling*-tapahtumana loogista puurakennetta ylöspäin. Lopulta tapahtuma päätyy määritettyyn *Label*-komponenttiin, jossa on esitelty *MouseUp*-tapahtuman tapahtumankäsittelijä. *Label*-komponentissa määritelty tapahtumankäsittelijä suoritetaan ja *MouseUp*-tapahtuma jatkaa kulkuaan puurakenteessa ylöspäin. Tapahtuman kulkua voidaan seurata toteuttamalla kaikille komponenteille oma *MouseUp*-tapahtuman tapahtumankäsittelijä. (MacDonald 2010, ss. 119-158).

Reititettyjen tapahtumien ansiosta näkymässä syntyvien tapahtumien käsittelijät voidaan toteuttaa vasta tietomallissa, joka asetetaan näkymän *DataContext*-määreeseen. Reititettyjen tapahtumien kulku päätyy lopulta *DataContext*-määreeseen asetettuun tietomalliin, jossa tarvittavat käsittelijät on toteutettu. Näin ollen näkymä ei sisällä mitään logiikkaa tai tapahtumankäsittelijöitä. Tämä mekanismi on välttämätön, jotta WPF-teknologian tiedon sidonta ja komennot toimivat. (MacDonald 2010).

3.2.4 Tiedon sitominen

Tiedon sitominen (eng. *Data Binding*) on WPF-teknologian tarjoama ominaisuus, jonka avulla voidaan toteuttaa tiedon välittäminen näkymien ja tietomallin välillä. WPF-teknologian tiedon sidonnan avulla näkymien ja tietomallin välille voidaan muodostaa vuorovaikutussuhde nopeasti ja tehokkaasti. (MacDonald 2010, ss 599-642: DataBinding 2011)

Esimerkki 4 kuvaa yksinkertaista tietomallia. Esimerkissä 4 on kuvattu *Contact*-luokka, joka toteuttaa yhden henkilön tiedot.

```
class Contact
{
   ...
    private string _firstName = String.Empty;
    public string FirstName
    {
        get { return _firstName; }
        set { _firstName = value; }
    }

    private string _surname = String.Empty;
    public string Surname
    {
        get { return _surname; }
        set { _surname = value; }
    }

    private string _phone = String.Empty;
    public string Phone
    {
        get { return _phone; }
        set { _phone = value; }
    }
}
```

Esimerkki 4. *Contact-luokan toteutus.*

Näkymän määrittämässä XAML-tiedostossa tiedon sidonta toteutetaan *Binding*-avainsanan avulla. Esimerkissä 5 on kuvattu yksinkertainen ikkuna, joka hyödyntää WPF-teknoligian tiedon sidontaa.

```
<Grid>
    ...
    <Label ... Content="First name:"/>
    <TextBox ... Text="{Binding Path=FirstName}"/>
    <Label ... Content="Last name:"/>
    <TextBox ... Text="{Binding Path=Surname}"/>
    <Label ... Content="Phone:"/>
    <TextBox ... Text="{Binding Path=Phone}"/>
</Grid>
```

Esimerkki 5. *Tiedon sidonta XAML-tiedostossa.*

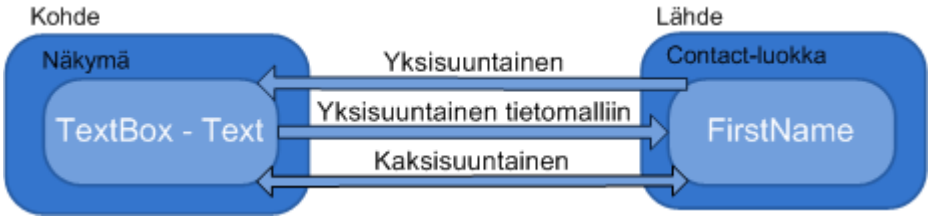
Esimerkissä 5 *TextBox*-komponenttien *Text*-kenttiin sidotaan edellä kuvatun *Contact*-luokan ominaisuudet. Tiedon sidonnan tärkeässä asemassa on näkymän *DataContext*-määre. *DataContext* kuvaa tietomallia, joka on näkymän käytettävänä. *DataContext* voidaan asettaa monelle eri tasolle. Koko näkymän käytössä oleva tietomalli voidaan asettaa esimerkin 6 mukaisesti.

```
public partial class ContactWindow : Window
{
    public ContactWindow()
    {
        InitializeComponent();
        this.DataContext = new Contact();
    }
}
```

Esimerkki 6. *Tietomallin asettaminen näkymälle.*

Esimerkissä 6 *DataContext*-määre määritellään näkymän XAML-kuvaustiedoston .cs-kooditiedostossa. Tällä menetelmällä tietomalli, joka on asetettu *DataContext*-kenttään, on koko näkymän käytettävissä. Tiedon sidonnan toteutuksessa hyödynnetään XAML-kuvauskielen loogista puurakennetta. *Binding*-avainsanalla määrittyä polkua lähdetään määrittämään kulkemalla loogista puurakennetta ylöspäin. Puurakennetta kuljetaan ylöspäin, kunnes löydetään ensimmäinen tietomalli, joka on asetettu näkymälle tai säiliölle. *DataContext* voidaan asettaa näkymän lisäksi myös käyttöliittymässä määritetyille säiliöille. Mikäli käyttöliittymässä on määritelty useita tietomalleja, tiedon sidonta hyödyntää ensimmäistä tietomallia, joka löytyy kuljettaessa loogista puurakennetta ylöspäin. WPF-teknologian tiedon sidonta liittää automaattisesti löydetyn tietomallin ominaisuuden määritetyn käyttöliittymäkomponentin ominaisuuteen. (MacDonald 2010, ss. 599-642: DataBinding 2011)

WPF-teknologian tiedon sidonta huolehtii tiedon välittämisen niin mallista näkymään kuin näkymästä takaisin tietomalliin. Tiedon sidontaan voidaan määritellä asetuksia, joiden avulla tiedon välitys voidaan toteuttaa halutulla tavalla. Tiedon sidontaa ja sen asetuksia havainnollistetaan kuvassa 13.



Kuva 13. *Tiedon sidonnan asetukset (Data Binding 2011).*

Kuvassa 13 on havainnollistettu tiedon sidonnan lähde ja kohde. Lähde on esimerkissä 4 kuvattu *Contact*-luokka, ja kohde on käyttöliittymäkomponentti ja sen ominaisuus. Kuvassa 13 on esitetty tiedon sidonta yhden ominaisuuden kohdalla. Tiedonvälityksen erilaiset asetukset jakautuvat viiteen vaihtoehtoon. (MacDonald 2010, ss. 249-264: DataBinding 2011)

- **Yksisuuntainen (OneWay):** Tieto päivitetään vain tietomallista (lähde) käyttöliittymään (kohde). Toiminnallisuus on hyödyllinen tilanteissa, jossa näkymässä näytetään käyttäjälle pelkästään tietoa eikä käyttäjälle haluta antaa mahdollisuutta tiedon muuttamiselle tai lisäämiselle.
- **Yhden kerran (OneTime):** Tieto päivitetään vain kerran sovelluksen käynnistyessä tai kun *DataContext*-määreeseen asetetaan uusi tietosisältö. Toiminnallisuutta voidaan hyödyntää, mikäli tiedetään, että tietosisältö on muuttumaton tai tietosisällön muutosta ei tahdota päivittää. Lataamalla tietosisältö vain kerran mahdollistetaan parempi suorituskyky, mikäli esitettävä tietomalli on suuri.
- **Kaksisuuntainen (TwoWay):** Tämä on yleisin asetus WPF-teknologian tiedon sidonnassa. Asetuksen mukaan tieto välitetään molempiin suuntiin, niin tietomallista näkymään, kuin näkymästä tietomalliin. Ominaisuus tukee käyttöliittymäsovelluksien yleisimmin käytettyä toiminnallisuutta, jossa näkymässä halutaan näyttää jotain tietoa ja antaa samalla käyttäjälle mahdollisuus muokata sitä.
- **Yksisuuntainen tietomalliin (OneWayToSource):** Tieto päivitetään vain näkymästä tietomalliin. Tämä toiminnallisuus on käytännöllinen esimerkiksi kaavakkeen täyttämisessä. Kaavakkeen täyttämisessä ei olla kiinnostuneita uuden tiedon näyttämisestä, vaan halutaan kerätä tietoa käyttäjältä.
- **Oletus (Default):** Mikäli tiedonvälityksen asetusta ei ole asetettu, käytetään oletusasetusta. Oletusasetus vaihtelee käytetyn käyttöliittymäkomponentin mukaan. Käyttöliittymäkomponentti, johon käyttäjä voi syöttää dataa, kuten *TextBox* ja *CheckBox* oletus asetus on kaksisuuntainen. Muille käyttöliittymäkomponenteille oletusasetus on yksisuuntainen.

Tiedon sidonnan eri asetukset voidaan asettaa *Mode*-määreen avulla esimerkin 7 mukaisesti.

```
<Grid>
     ...
     <TextBox Text="{Binding Path=FirstName, Mode=TwoWay}"/>
     ...
 </Grid>
```

Esimerkki 7. *Tiedon sidonnan Mode-määreen asettaminen.*

Tiedon sidonta huolehtii tiedon välittämisen näkymän ja tietomallin välillä, mutta ohjelmoijan tulee määritellä milloin tieto on muuttunut tai milloin muutos tahdotaan välittää eteenpäin. Näkymässä tiedon muutoksesta voidaan ilmoittaa *UpdateSourceTrigger*-määreen avulla esimerkin 8 mukaisesti. Näkymässä tapahtuvan ilmoituksen tiedon muutoksesta voi toteuttaa neljällä eri tavalla. (MacDonald 2010, ss 249-264: DataBinding 2011)

```
<Grid>
    ...
    <TextBox Text="{Binding Path=FirstName, Mode=TwoWay,
    UpdateSourceTrigger=PropertyChanged}"/>
    ...
</Grid>
```

Esimerkki 8. *Tiedon sidonnan UpdateSourceTrigger-määreen asettaminen.*

- **Tieto muuttunut (PropertyChanged):** Tämä vaihtoehto on yleisin vaihtoehto. Näkymä välittää tiedon muutoksesta heti, kun tieto muuttuu.
- **Kohdistuksen poistuminen (LostFocus):** Ilmoitus tiedon muuttumisesta välitetään silloin, kun käyttäjä siirtää kohdistuksen pois kyseisestä käyttöliittymäkomponentista.
- **Eksplisiittinen (Explicit):** Ilmoitus tiedon muuttumisesta ei ole automatisoitu. Ohjelmoijan tulee kutsua itse *BindingExpression.UpdateSource()*-metodia, kun haluaa muutoksien päivittyvän.
- **Oletus (Default):** Jokaiselle käyttöliittymäkomponentille on asetettu oma oletusarvonsa. Yleensä oletusarvo on listan ensimmäinen vaihtoehto (tieto muuttunut), mutta *TextBox*-komponentilla muutokset välitetään oletusarvoisesti kohdistuksen poistuessa tekstikentästä.

Tietomallin muutos näkymälle toteutetaan *InotifyPropertyChanged*-rajapinnan avulla. Esimerkissä 9 on toteutettu tarvittava rajapinta *Contact*-luokalle.

```
class Contact : INotifyPropertyChanged
{
   public event PropertyChangedEventHandler PropertyChanged;
   public void OnPropertyChanged(PropertyChangedEventArgs e)
   {
      if (PropertyChanged != null)
         PropertyChanged(this, e);
   }
   ...
}
```

Esimerkki 9. *Tiedon sidonnan INotifyPropertyChanged-rajapinnan toteutus tietomallissa.*

Näkymälle tietomallin muutoksesta ilmoitetaan *PropertyChanged*-tapahtuman avulla. Tapahtuma on ominaisuuskohtainen, joten jokaiselle ominaisuudelle, joka on sidottu käyttöliittymän komponenttiin, tulee toteuttaa oma tapahtuman laukaisija. Tapahtuman parametreihin asetetaan ominaisuuden nimi, joka on muuttunut. Näkymä tunnistaa ominaisuuden nimen perusteella, mikä tieto on muuttunut ja suorittaa tiedon päivityksen. Tapahtuma voidaan laukaista kootusti tietomallin päivityksen yhteydessä, jolloin kaikkien ominaisuuksien *PropertyChanged*-tapahtuma suoritetaan peräkkäin tai esimerkin 10 mukaisesti aina kun ominaisuudelle asetetaan uusi arvo.

```
private string _firstName = String.Empty;
public string FirstName
{
   get { return _firstName; }
   set
   {
      _firstName = value;
      OnPropertyChanged(new PropertyChangedEventArgs("FirstName"));
   }
}
```

Esimerkki 10. *Tietomallin muutoksen ilmoittaminen PropertyChanged-tapahtumalla.*

WPF-teknologian tiedon sidonta voidaan toteuttaa myös pelkästään näkymässä kahden eri käyttöliittymäkomponentin välille (MacDonald 2010, s. 249). Tiedon sidonta ja käyttöliittymän määritys XAML-kuvauskielellä ovat WPF-teknologian tärkeimpiä työkaluja, jotka mahdollistavat malli-näkymä-arkkitehtuureilta vaadittavan näkymän erottamisen sovelluslogiikasta. XAML-kuvauskielen avulla tiedon sidonta on helppo ja nopea toteuttaa. Tiedon sidontaa hyödynnetään myös tietomallin toimintojen sitomiseen käyttöliittymän komponentteihin, josta enemmän kappaleessa 3.2.7 Toiminnon suorittaminen.

3.2.5 Tiedon validointi

Käyttäjän syöttämän tiedon validointi on yleinen toiminnallisuus käyttöliittymäsovelluksissa. Tiedon validointi voidaan toteuttaa monella eri tasolla, ja kaikissa toteutusratkaisuissa on hyvät ja huonot puolensa.

Tiedon validointi voidaan toteuttaa suoraan käyttöliittymäkomponentissa. Tässä tapauksessa käyttöliittymäkomponenttiin voidaan toteuttaa mekanismi, joka hyväksyy esimerkiksi vain tiettyjen painikkeiden painallukset. Ongelmana tässä ratkaisussa on toteutuksen uudelleen käyttäminen. Mikäli saman tiedon käsittelyyn toteutetaan toinen näkymä, tulee sama toteutus toteuttaa myös sitä kautta tietoa käsitteleville komponenteille.

Tietokantaa käytettäessä tiedon validointi voidaan toteuttaa tietokantaan. Tietokannalle voidaan toteuttaa määrityksiä, jotka kuvaavat minkälaista tietoa tauluihin voidaan tallentaa. Ongelmana kuitenkin on vanhan tiedon palauttaminen. Ilmoitus virheellisestä tiedosta saadaan vasta, kun tietoa yritetään tallentaa tietokantaan. Kun kantaan yritetään lisätä laitonta tietoa, tietokannan päivitystransaktio epäonnistuu, jolloin tietokanta tulisi voida palauttaa vanhaan tunnettuun tilaan. Samassa transaktiossa on voitu tallentaa jo jotain muuta tietoa, eikä tietokannan tarkkaa tilaa välttämättä voida selvittää. Operaatioiden jälkeen tietokannan eheys on vaikea todentaa.

WPF-teknologia tarjoaa tiedon validointiin ratkaisun, jossa tiedon validointi toteutetaan tietomallissa ennen tiedon tallennusta tietokantaan. WPF-teknoligian tiedon validointi on toteutettu tiedon sitomisen yhteyteen. Tiedon sidonnalle voidaan asettaa *Vali-*

datesOnDataErrors-muuttuja todeksi, jolloin WPF-teknoligian tiedon sidonta tarkistaa tiedon laillisuuden *IDataErrorInfo*-rajapinnan mukaisesti. Esimerkissä 11 tiedon validointi on otettu käyttöön *Contact*-luokan *Phone*-muuttujalle.

```
<Grid>
    ...
    <TextBox Text="{Binding Path=Phone, Mode=TwoWay,
     UpdateSourceTrigger=PropertyChanged, ValidatesOnDataErrors=True}"/>
    ...
</Grid>
```

Esimerkki 11. *ValidatesOnDataErrors-määreen asettaminen näkymässä.*

Varsinainen logiikka, joka tarkistaa tiedon laillisuuden toteutetaan tietomalliin. Tietomallin tulee toteuttaa *IDataErrorInfo*-rajapinta, jonka avulla näkymälle viestitään tiedon laillisuudesta. Esimerkissä 12 on toteutettu *IDataErrorInfo*-rajapinta *Contact*-luokalle. Esimerkkitoteutus tarkistaa, että annettu puhelinnumero sisältää vain numeroita.

```
class Contact : IDataErrorInfo
{
...

   public string Error
   {
      get { return null; }
   }

   public string this[string propertyName]
   {
      get
      {
         if (propertyName == "Phone")
         {
            if (this.Phone != null)
            {
               bool isValid = false;
               Regex regex = new Regex("^[0-9]+$");

               if (regex.IsMatch(this.Phone))
                  isValid = true;

               if (!isValid)
                  return "Only numbers are allowed in phone number!";
            }
         }
        return null;
    }
}
```

Esimerkki 12. *IDataErrorInfo-rajapinnan toteutus tietomallissa.*

IDataErrorInfo-rajapinnasta pitää toteuttaa tietomallille merkkijono indeksointi. Metodi saa parametrina validoitavan ominaisuuden nimen. Ominaisuuden nimen avulla voidaan selvittää, minkä tiedon validointia näkymä pyytää. Jokaiselle ominaisuudelle, jolle halutaan toteuttaa tiedon validointi, toteutetaan oma validointilogiikka. Indeksointi palauttaa *string*-merkkijonon. Mikäli validoitava muuttuja on laillinen, palautetaan arvo *null*. Mikäli tieto on laitonta, palautetaan merkkijono. Merkkijono toimii samalla virheilmoituksena, joka voidaan esittää sovelluksen käyttäjälle näkymässä. WPF-teknologian tiedon sidonta kutsuu tiedon validointia automaattisesti aina, kun tieto muuttuu. *IDataErrorInfo*-rajapinta pitää sisällään myös *Error*-metodin, mutta tämä metodi ei ole käytössä WPF-käyttöliittymäkirjastossa, joten se palauttaa aina arvon *null*.

Näkymässä virheilmoitus voidaan näyttää monella eri tavalla. Oletuksena käyttöliittymäkomponentin reunat muuttuvat punaiseksi mikäli käyttäjän syöttämä tieto on laitonta. Esimerkki virheilmoituksen näyttämisestä on esitetty kappaleessa 3.3.3 AddContactUserControl-näkymä.

3.2.6 Tiedon esittäminen

XAML-kuvauskielellä määritelty käyttöliittymä kuvaa pelkästään, kuinka sovelluksen tieto esitetään käyttöliittymässä. WPF-käyttöliittymäkirjaston avulla toteutetut käyttöliittymät rakentuvat sen mukaisesti, että käyttöliittymälle annetaan jokin tietorakenne, joka pitää sisällään esitettävän tiedon, ja käyttöliittymässä kuvataan, minkälaisilla menetelmillä ja komponenteilla annettu tietomalli esitetään käyttöliittymässä. Tärkeässä asemassa tiedon esittämisessä käyttöliittymässä on tiedon esitysmallit (eng. *Data Template*). (MacDonald 2010, ss. 599-689)

WPF-teknologian esitysmallien avulla esimerkiksi erilaisille listakomponenteille voidaan määritellä malli, jonka mukaan kaikki listan alkiot esitetään määrätyllä tavalla. Esimerkissä 13 on määritelty *ListView*-komponentti, jonka tietosisällöksi asetetaan lista sivulla 26 esimerkissä 4 esiteltyjä *Contact*-luokan instansseja.

```
<ListView  ItemsSource="{Binding ElementName=This, Path=ListOfContacts}"/>
```

Esimerkki 13. *ListView-komponenttin määrittäminen.*

Ilman lisämäärityksiä *ListView*-komponentti esittää sille asetetun tietosisällön alkioista tietoa *ToString()*-metodia käyttäen. Oletusarvoisesti tietomallin alkioiden *ToString()*-metodi palauttaa luokan nimen.

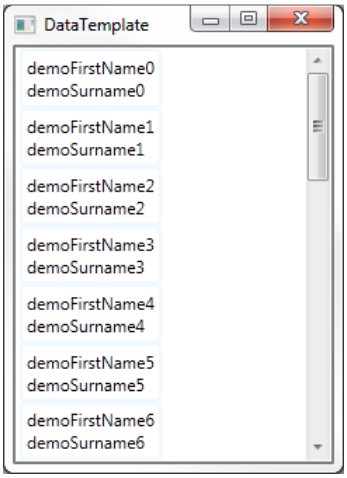
Tiedon esittämiseen listassa voidaan vaikuttaa kolmella eri tavalla. Ensimmäinen tapa on toteuttaa tietomallin luokalle *ToString()*-metodi, jonka avulla palautetaan jokin haluttu merkkijono, joka kuvaa tietomallin sisältöä. Toinen vaihtoehto on lisätä tiedon sidonnan määrittämiseen *DisplayMemberPath*-määre, jonka avulla listassa voidaan esittää jokin yksi ominaisuus tietomallin instanssista. Kolmas ja monipuolisin vaihtoehto on toteuttaa tietomallille tiedon esitysmalli. Esitysmallin avulla tietomalli voidaan esittää

halutulla tavalla listassa. *ListView*-komponentin esitysmalli määritellään *ItemTemplate*-määreellä. *ItemTemplate*-määreelle voidaan asettaa mikä tahansa määritelty esitysmalli (*DataTemplate*). (MacDonald 2010, ss. 599-689). Esimerkissä 14 on kuvattu esitysmallin määrittäminen *ListView*-komponentille.

```
<ListView  ItemsSource="{Binding ElementName=This, Path=ListOfContacts}">
   <ListView.ItemTemplate>
      <DataTemplate>
         <Border BorderThickness="2" BorderBrush="AliceBlue" CornerRadius="4">
            <Grid Margin="3">
               <Grid.RowDefinitions>
                  <RowDefinition Height="Auto"/>
                  <RowDefinition Height="Auto"/>
               </Grid.RowDefinitions>
               <TextBlock Grid.Row="0" Text="{Binding Path=FirstName}"/>
               <TextBlock Grid.Row="1" Text="{Binding Path=Surname}"/>
            </Grid>
         </Border>
      </DataTemplate>
   </ListView.ItemTemplate>
</ListView>
```

Esimerkki 14. *Esitysmallin määrittäminen ListView-komponentille.*

Esimerkissä 14 on kuvattu yksinkertainen esitysmalli, joka esittää annetusta tietomallista *FirstName*- ja *Surname*-ominaisuudet. Tietomalli voi pitää sisällään myös muitakin ominaisuuksia, mutta tässä näkymässä näytetään vain nämä ominaisuudet. XAML-kuvauskielen ansiosta esitysmallissa voidaan määritellä mitä tahansa WPF-teknologian käyttöliittymäkomponentteja. Esimerkin 14 mukainen näkymä on esitetty kuvassa 14.



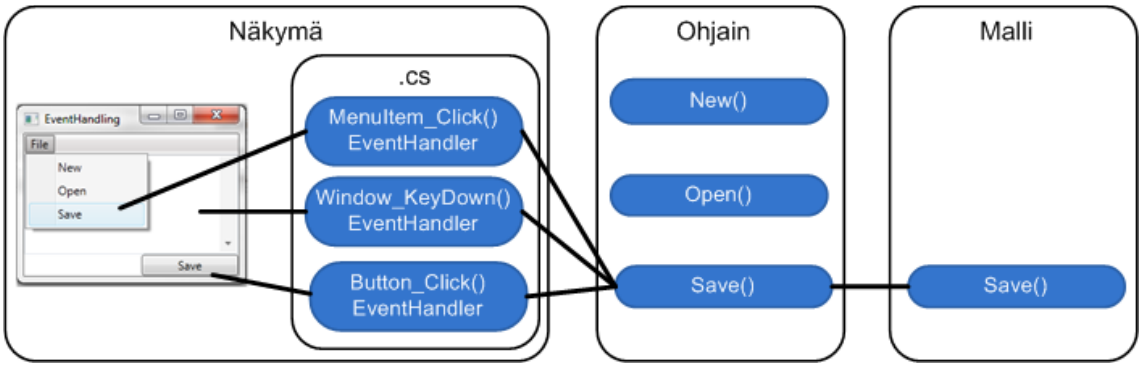
Kuva 14. *Esitysmallia hyödyntävä ListView-komponentti.*

3.2.7 Toiminnon suorittaminen

WPF-teknologiassa toimintojen suorittaminen toteutetaan komentojen avulla. Komennot (eng. *Commands*) ovat WPF-teknoligian tarjoama syötteiden käsittely mekanismi. Komentojen tarkoitus on erottaa toiminnon lähde sen käsittelijästä. Käytännössä tämä tarkoittaa sitä, että sama toiminnallisuus voidaan suorittaa usean eri lähteen avulla. Kopioi ja liitä -ominaisuus on esimerkki tämänkaltaisesta toiminnallisuudesta. Kopioi ja liitä -ominaisuudessa on tyypillistä, että yksi yksittäinen toiminto voidaan laukaista useasta eri lähteestä. Käyttäjälle voidaan tarjota esimerkiksi valikon toiminto, painike ja pikanäppäinyhdistelmä, joiden avulla sama toiminnallisuus voidaan suorittaa. (Commanding Overview 2011.)

WPF-teknologian komennot tarjoavat MVVM-suunnittelumallin toteutukseen tärkeän työkalun, jonka avulla käyttöliittymästä saadaan toteutettua itsenäinen kokonaisuus, joka ei ole sidoksissa järjestelmän sovelluslogiikkaan. Komentojen ansiosta järjestelmään ei tarvitse kirjoittaa toistuvia tapahtumankäsittelijöitä. (Commanding Overview 2011; MacDonald 2010, ss. 265-292)

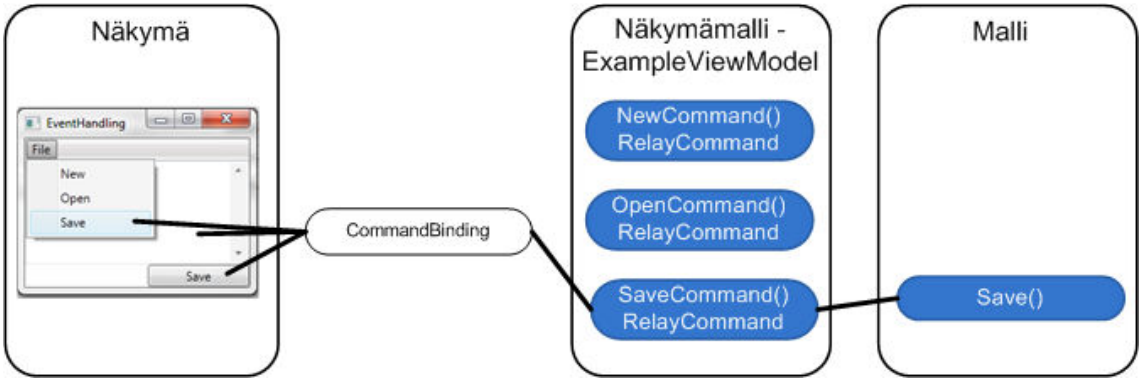
Hyvin suunnitellussa käyttöliittymäsovelluksessa on olemassa komponentti, joka pitää sisällään sovelluksen tarjoamat toiminnallisuudet. Otetaan käsittelyyn esimerkiksi tiedon tallennus operaatio. Kuva 15 esittää tilannetta, kuinka tallenna-operaatio suoritetaan perinteisesti tapahtumankäsittelijöiden avulla. (MacDonald 2010, ss. 265-292.)



Kuva 15: *Tallenna-operaatio tapahtumankäsittelijöiden avulla (MacDonald 2010, s.265).*

Kuvasta 15 nähdään, kuinka jokaiselle käyttöliittymän komponentille on toteutettava oma tapahtumankäsittelijä, joka kytketään haluttuun operaatioon. Tämä on oikea tapa toteuttaa järjestelmiä, mutta se aiheuttaa turhaa työtä tapahtumankäsittelijöiden kirjoittamiselle. Lopulta kaikki tapahtumankäsittelijät kutsuvat samaa operaatiota. Tapahtumankäsittelijöiden avulla tämä toiminnallisuus voidaan toteuttaa, mutta toiminnallisuuden toteutus hankaloituu, mikäli järjestelmään pitää toteuttaa ehtoja, milloin tallenna-operaatio saadaan suorittaa. Jokaiselle käyttöliittymäkomponentille tulee toteuttaa oma käsittelijänsä, joka vaihtaa komponentin *IsEnabled*-ominaisuutta. Tämän tapaisen tilakoneen toteuttaminen voi muodostua monimutkaiseksi ja vaikeaksi ylläpitää. (MacDonald 2010, ss. 265-292.)

Smith (2009) esitteli *RelayCommand*-luokan, joka hyödyntää WPF-teknologian *ICommand*-rajapintaa. *RelayCommand*-luokka toteuttaa mekanismin, minkä avulla käyttöliittymässä tapahtuvia toiminnallisuuksia voidaan sitoa näkymämallin tarjoamiin operaatioihin. *RelayCommand*-luokan avulla kuvan 15 ongelma voidaan esittää kuvan 16 avulla.



Kuva 16: *Tallenna-operaatio komentojen avulla (MacDonald 2010, s. 266).*

Kuvasta 16 nähdään, kuinka kaikki käyttöliittymän komponentit, joiden tarkoitus on suorittaa sama operaatio, sidotaan yhteen näkymämallin komentoon. *RelayCommand*-luokka hyödyntää WPF-teknologian komentomallia. WPF-teknologian komentomalli koostuu seuraavista neljästä kokonaisuudesta (MacDonald 2010, s. 267):

- **Komento (*Command*):** Komento tarjoaa rajapinnan, jonka avulla järjestelmän operaatio voidaan suorittaa. Lisäksi komento pitää sisällään tietoa siitä, milloin komento voidaan suorittaa. WPF-teknoligian komentomallin ansiosta käyttöliittymäkomponentit muuttavat *IsEnabled*-tilaansa komennon tilan mukaan.
- **Komennon sitominen (*Command Binding*):** *CommandBinding*-luokka tarjoaa mekanismin, minkä avulla komento voidaan sitoa näkymämallin metodiin.
- **Komennon lähde (*Command Source*):** Komennon lähde on käyttöliittymä komponentti, joka laukaisee komennon. Komento voidaan laukaista myös kutsumalla suoraa komentoa, jolloin komennon ei tarvitse olla kytkettynä käyttöliittymäkomponenttiin.
- **Komennon kohde (*Command Target*):** Komennon kohde on MVVM-suunnittelumallin tapauksessa näkymämallissa toteutettu metodi.

RelayCommand-luokan (Smith 2009) toteutus on esitetty liitteessä 10. *RelayCommand-luokka* on periytetty WPF-teknologian *ICommand*-rajapinnasta. *ICommand*-rajapinta tarjoaa *Execute* ja *CanExecute*-metodit sekä *CanExecuteChanged*-tapahtuman esimerkin 15 mukaisesti.

```
public interface ICommand
{
   void Execute (object parameter);
   bool CanExecute (object parameter);
   event EventHandler CanExecuteChanged;
}
```

Esimerkki 15. *ICommand-rajapinta. (ICommand 2011)*

Tallenna-operaation toteuttaminen *RelayCommand*-luokan avulla on esitelty esimerkissä 16.

```
public class ExampleViewModel
{
  ...
   private RelayCommand _saveCommand;
   public ICommand SaveCommand
   {
      get
      {
         if (_saveCommand == null)
         {
            _saveCommand = new RelayCommand(
               param => this.Save(),
               param => this.CanSave
               );
         }
         return _saveCommand;
      }
   }

   private void Save()
   {
      ...
   }

   private bool CanSave
   {
      ...
   }
   ...
}
```

Esimerkki 16. *SaveCommand-komennon toteuttaminen RelayCommand-luokan avulla.*

RelayCommand-luokka hyödyntää delegaatteja luokan rakentajassa. Menetelmän avulla käyttäjä voi määritellä omat toiminnallisuutensa *Execute* ja *CanExecute*-metodeille. Esimerkissä 16 *Save*-metodi toteuttaa toiminnallisuuden, mikä tarvitaan

tiedon tallentamiselle, ja *CanSave*-ominaisuus toteuttaa logiikan milloin sovellus on siinä tilassa, että tieto voidaan tallentaa. *RelayCommand*-luokan toteutuksessa, liitteessä 10 rivillä 50, kytketään *CanExecuteChanged*-tapahtuma WPF-teknoligian *CommandManager*-luokan *RequerySuggested*-tapahtumaan. Kytkennän ansiosta WPF-teknologian komentojen käsittelijä kutsuu *CanExecuteChanged*-tapahtumaa, joka aiheuttaa *RelayCommand*-luokan *CanExecute*-metodin kutsumisen. Tämä mekanismi toteuttaa toiminnallisuuden, jonka avulla komentoon liitetyt käyttöliittymäkomponentit osaavat asettaa oman *IsEnabled*-muuttujan tilan oikein sen mukaan, voidaanko komentoa suorittaa vai ei.

WPF-teknologian komentojen käsittelijä laukaisee automaattisesti tapahtumia, jotka aiheuttavat *CanExecute*-metodien suorittamisen. Kaikkien käyttöliittymään kytkettyjen komentojen *CanExecute*-metodeja kutsutaan useasti. Tämän takia *CanExecute*-metodien toteutus on suositeltavaa olla nopea. WPF-teknologian komentojen käsittelijä laukaisee operaation satunnaisesti silloin, kun se katsoo päivittämisen olevan tarpeellista. Operaatio laukaistaan muun muassa, kun käyttöliittymän kohdistus muuttuu tai jokin komento suoritetaan. WPF-teknologian komentojen käsittelijä kutsuu päivitysoperaatiota usein, mikäli esiintyy tilanne, jossa päivitysoperaatio tulisi suorittaa, voidaan operaatio laukaista käsin kutsumalla komennon *CanExecute*-metodia. Näkymämalliin voidaan toteuttaa myös *RelayCommand*-komento, joka ei toteuta *CanExecute*-metodia. Tässä tapauksessa komento voidaan aina suorittaa. (MacDonald 2010, s. 265-292; Smith 2009).

Käyttöliittymässä komennon sitominen käyttöliittymäkomponenttiin on yksinkertaista. Esimerkissä 17 *SaveCommand*-komento sidotaan näppäimistön CTRL+S-painikkeeseen pikanäppäimenä sekä käyttöliittymän *Save*-painikkeeseen.

```
...
<Window.InputBindings>
   <KeyBinding Command="{Binding Path=SaveCommand}" Gesture="CTRL+S"/>
</Window.InputBindings>
...

<Button ... Content="Save" Command="{Binding Path=SaveCommand}" />
...
```

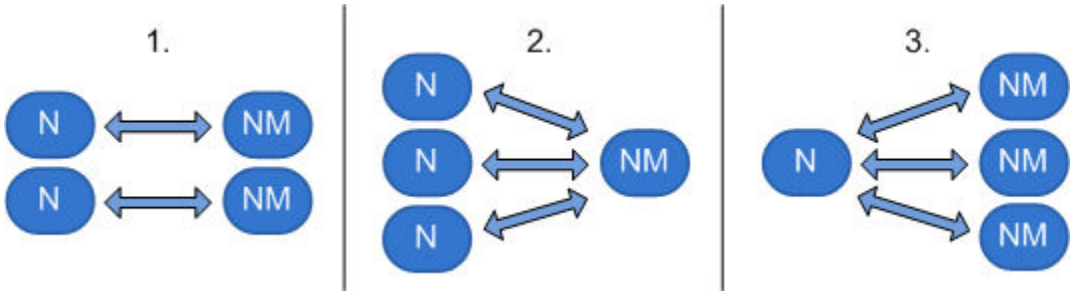
Esimerkki 17. *Komennon sitominen käyttöliittymään.*

Esimerkin 17 *Command*-ominaisuuden määrittely luo uuden *CommandBinding*-olion. Kun käyttäjä painaa *Save*-painiketta, alkaa tapahtumaketju, jossa komentoon sidottu tapahtuma vyöryy *bubbling*-ominaisuudelle puurakennetta ylöspäin. Tapahtuma etsii *SaveCommand*-komennon toteutusta. Komennon sitominen toimii, mikäli näkymän *DataContext*-ominaisuuteen on asetettu tietomalli, joka toteuttaa kyseisen komennon.

3.3 Puhelinluettelosovelluksen toteutus MVVM-suunnittelumallilla

3.3.1 Osien väliset riippuvuudet

Kappaleessa 2.3.4 esiteltiin MVVM-suunnittelumallin rakenne ja eri osien vastuualueet. Kappaleessa kuvattiin, kuinka MVVM koostuu näkymästä, näkymämallista sekä mallista. MVVM-suunnittelumallin mukaisessa toteutuksessa osien välille voidaan toteuttaa erilaisia lukumääräsuhteita. Anderson (2010, ss. 378-380) esittelee kolme erilaista kokoonpanoa suunnittelumallin osien välille. Kolme erilaista kokoonpanoa on esitelty kuvassa 17.



Kuva 17: *Näkymien ja näkymämallien väliset kolme kokoonpanoa.*

Ensimmäisessä kokoonpanossa sovelluksen näkymät ja näkymämallit muodostavat pareja. Jokaiselle sovelluksen näkymälle toteutetaan yksi näkymämalli, joka tarjoaa näkymälle sen tarvitseman tiedon ja toiminnallisuuden. Andersonin mukaan ensimmäisen kokoonpanon mukainen toteutus on perinteinen tapa toteuttaa MVVM-suunnittelumallin mukainen sovellus. Kokoonpano pitää sovelluksen rakenteen yksinkertaisena ja sen avulla suunnittelumalli on helppo ymmärtää ja oppia. (Anderson 2010, ss. 378-380.)

Seuraava kokoonpano kuvaa tilannetta, jossa sovelluksen yksi näkymämalli tarjoaa palveluita usealle näkymälle. Kokoonpano on hyödyllinen tilanteessa, jossa useampi näkymä käsittelee samaa tietoa. Anderson kuvaa ohjatun asennuksen (eng. *wizard*) olevan hyvä esimerkki tällaisesta tilanteesta. Ohjatussa asennuksessa käyttäjälle esitellään jokin kokonaisuus useassa eri vaiheessa. Käyttäjä käy läpi kaikki tarvittavat vaiheet, ja lopulta eri vaiheet muodostavat yhden suuren kokonaisuuden. Ohjatun asennuksen toteutuksessa yhteen näkymämalliin toteutetaan kaikki tarvittava vaiheet. Tämän jälkeen toteutetaan näkymät, jotka mahdollistavat eri vaiheiden käytön. Toteutus pysyy yksinkertaisena, koska kaikkien näkymien käsittelemä tieto on koottu samaan paikkaan. (Anderson 2010, ss. 378-380.)

Viimeisessä, eli kolmannessa kokoonpanossa yksi näkymä hyödyntää usean eri näkymämallin tarjoamia palveluita. Perinteisesti näkymämalli toteutetaan palvelemaan jotain tiettyä näkymää, jolloin näkymämalli kapseloi tarvittavat tiedot mallista niin, että näkymän on helppo käyttää samanaikaisesti useaa luokkaa mallista. Kolmas kokoonpa-

no kääntää ajatusmallin päinvastoin. Kolmannen kokoonpanon mukaisesti näkymämalli kapseloi jonkin tietyn kokonaisuuden tietomallista. Näin ollen näkymä käyttää monta eri näkymämallia tiedon esittämiseen näkymässä. Andersonin mukaan kolmannen kokoonpanon ongelmana on, että kokoonpano kasvattaa näkymämallien lukumäärää ja lisää koodirivejä, joita tarvitaan näkymämallien rakentamiseen. (Anderson 2010, ss. 378-380.)

MVVM-suunnittelumalli ei rajoita näkymien ja näkymämallien välisiä lukumääräsuhteita. Puhelinluettelosovelluksessa eri osien lukumääräsuhteet on toteutettu kuvan 18 mukaisesti.

Kuva 18: *MVVM-suunnittelumallin osien väliset suhteet puhelinluettelosovelluksessa.*

Yhteyshenkilön lisäämisessä on hyödynnetty ensimmäisen kokoonpanon mukaista ratkaisua. *AddContactUserControl*-näkymä on sidottu käyttämään *ContactViewModel*-näkymämallin palveluita. Puhelinluettelosovelluksen malli on yksinkertainen, joten *ContactViewModel*-näkymämalli toteuttaa toiminnallisuuden, jonka avulla sovelluksen mallia eli yhteystietoja voidaan käsitellä.

SearchContactUserControl-näkymän palvelut on toteutettu *SearchContactViewModel*-näkymämalliin. Näkymämalli hyödyntää *ContactViewModel*-näkymämallia yhteyshenkilöiden tiedon esittämiseen. *SearchContactViewModel*-näkymämalli ei ole suoraan yhteydessä malliin vaan käyttää mallin palveluita *ContactViewModel*-näkymämallin tarjoamana. Näin ollen näkymä, jossa yhteyshenkilöiden tietoja esitetään, on sidottu sovelluksen molempiin näkymämalleihin.

3.3.2 AddressBookMainWindow-näkymä

Puhelinluettelosovelluksen päänäkymän toteutus on esitelty liitteessä 1. Näkymän XAML-tiedostossa käytetään sovelluksen kahta alinäkymää. Näkymien vaihtamiseen on toteutettu valikko. Valikon painikkeiden tapahtumankäsittelijät on toteutettu näkymän .cs-kooditiedostoon. MVVM-suunnittelumallin ohjeistus on, että .cs-kooditiedostossa ei saisi olla toteutusta. Monimutkaisuuden välttämiseksi näkymien vaihtoon liittyvää toiminnallisuutta voidaan toteuttaa näkymän .cs-kooditiedostoon. Näkymän vaihtamiseen liittyvien tapahtumankäsittelijöiden toteutus .cs-kooditiedostoon voidaan pitää perusteltuna, koska toteutettu toiminnallisuus ei sisällä sovelluslogiikkaa, jota jokin muu sovelluksen osa voisi tarvita.

WPF-sovelluksissa päänäkymän määritys voidaan tehdä muun muassa projektin *App.xaml*-tiedostossa. Tiedostoon lisätään *StartupUri*-määre, jonka avulla voidaan mää-

ritellä haluttu XAML-tiedosto, jonka halutaan olevan ensimmäinen näkymä sovelluksen käynnistyessä. Esimerkissä 18 on esitelty puhelinluettelosovelluksen päänäkymän määrittäminen *App.xaml*-tiedostossa.

```
<Application x:Class="AddressBook.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="View\AddressBookMainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

Esimerkki 18. *Puhelinluettelosovelluksen päänäkymän määrittäminen.*

3.3.3 AddContactUserControl-näkymä

Yhteyshenkilöiden lisäämiseen tarkoitettu näkymä on *AddContactUserControl*-näkymä. Näkymän toteutus on esitelty liitteessä 2. Näkymän muodostama käyttöliittymä esiteltiin sivulla 17 kuvassa 8.

Näkymässä on kuvattu tekstikenttiä, joiden avulla käyttäjä voi asettaa uuden yhteyshenkilön tiedot. Kaikki tekstikentät ovat sidottu *ContactViewModel*-näkymämallin ominaisuuksiin tiedon sidonnan avulla. Syötettävälle tiedolle on toteutettu tiedon validointi. Tiedon validointi on toteutettu *ContactViewModel*-näkymämallissa kappaleessa 3.2.5 esitetyn *IDataErrorInfo*-rajapinnan avulla. Käyttäjälle esitetään virheilmoitus laittomasta tiedosta. Virheilmoituksen esittämiseen käytetään *ContentPresenter*-luokkaa esimerkin 19 mukaisesti.

```
<TextBox ... x:Name="textBoxFirstName" Text="{Binding Path=FirstName,
   ValidatesOnDataErrors=True, UpdateSourceTrigger=PropertyChanged}"/>

<ContentPresenter ... Content="{Binding ElementName=textBoxFirstName,
   Path=(Validation.Errors).CurrentItem.ErrorContent}"
   TextBlock.Foreground="Red" />
```

Esimerkki 19. *Virheilmoituksen esittäminen käyttäen ContentPresenter-luokkaa.*

Uuden yhteyshenkilön lisäys on toteutettu *Add Contact*-painikkeella. Painike on sidottu *AddContactCommand*-komentoon, joka on toteutettu *ContactViewModel*-näkymämallissa.

3.3.4 SearchContactUserControl-näkymä

Kaikki sovelluksen yhteyshenkilöt listataan *SearchContactUserControl*-näkymässä. Näkymän toteutus on esitelty liitteessä 3. Näkymän muodostama käyttöliittymä esiteltiin sivulla 17 kuvassa 7.

Yhteyshenkilöitä voidaan etsiä sovelluksesta syöttämällä haluttu merkkijonon *SearchTextBox*-komponenttiin. *SearchTextBox*-komponentti on sidottu *SearchContactViewModel*-näkymämallin *SearchKey*-ominaisuuteen. Valittujen asetuksien ansiosta merkkijonon muutos välitetään heti näkymämallille. Tämän seurauksena yhteyshenkilöiden lista päivittyy aina, kun merkkijono muuttuu. Lista yhteyshenkilöistä voidaan päivittää myös painamalla *Search*-painiketta. Painike on sidottu *SearchContactViewModel*-näkymämallin *SearchCommand*-komentoon. Komennon toteutus ottaa huomioon annetun merkkijonon ja hakee yhteyshenkilöitä merkkijonon perusteella.

Puhelinluettelosovelluksessa haetut yhteyshenkilöt näytetään listassa. Lista on toteutettu *ListView*-komponentilla, jonka XAML-määritys on kuvattu esimerkissä 20.

```
<ListView ... Name="ContactsListView"
   ItemContainerStyle="{StaticResource alternatingListViewItemStyle}"
   AlternationCount="2"
   ItemsSource="{Binding Path=ListOfSearchedContacts}"
   ItemTemplate="{StaticResource contactListViewDataTemplate}"
   SelectedItem="{Binding Path=SelectedContact}"
   HorizontalContentAlignment="Stretch"/>
```

Esimerkki 20. *Yhteyshenkilöiden listaus ListView-komponentilla.*

ListView-komponentti käyttää tiedon lähteenä näkymämallin *ListOfSearchedContacts*-ominaisuutta. Ominaisuus palauttaa listan *ContactViewModel*-olioita, eli samoja olioita, joita käytettiin *AddContactUserControl*-näkymässä, jossa yhteyshenkilöitä voitiin lisätä. *ItemContainerStyle*- ja *AlternationCount*-määreiden avulla yhteyshenkilöiden listan taustavärit saadaan vaihdettua riveittäin. Yhteyshenkilön esittämiseen listassa hyödynnetään *contactListViewDataTemplate*-esitysmallia. Käytettävä esitysmalli on määritelty liitteessä 3 riveillä 9-31. Esitysmallin ansiosta listassa näytetään yhteyshenkilöstä vain etu- ja sukunimi sekä puhelinnumero. Lisäksi esitysmalli määrittelee listan alkioille siniset kehykset.

Valittu yhteyshenkilö tallennetaan näkymämallin *SelectedContact*-ominaisuuteen. Näkymän alareunaan on toteutettu *Information*-alue, joka esittää valitun yhteyshenkilön kaikki tiedot. Tiedot esitetään *GroupBox*-komponentin sisällä. Koska koko näkymä on sidottu samaan *SearchContactViewModel*-näkymämalliin, *Information*-alueessa näytettävä tieto voidaan sitoa *SelectedContact*-ominaisuuteen. Toteutus on esitelty liitteessä 3 riveillä 66-94.

Näkymien toteutuksista liitteistä 1, 2 ja 3 voidaan havaita, että WPF-teknologialla toteutetut näkymät määrittelevät vain tavan, kuinka sovelluksen tietoa halutaan näyttää käyttöliittymässä. WPF-teknoligian tiedon sidonnan ja komentojen ansiosta varsinainen tieto ja toiminnallisuus voidaan eriyttää näkymästä ja toteuttaa näkymämalleissa.

3.3.5 BaseViewModel-näkymämalli

Puhelinluettelosovelluksessa kaikki näkymämallit ovat periytettyjä abstraktista *BaseViewModel*-luokasta. *BaseViewModel*-luokka on näkymämalli, jossa on toteutettu kaikille näkymämalleille yhteisiä toiminnallisuuksia. Puhelinluettelosovelluksessa *BaseViewModel*-näkymämallissa on toteutettu *PorpertyChanged*-tapahtuman määrittäminen. Tapahtuman avulla ilmoitetaan ominaisuuden muutoksesta näkymälle tai näkymämallia käyttävälle komponentille. *BaseViewModel*-näkymämallin toteutus on esitelty liitteessä 4.

3.3.6 SearchContactViewModel-näkymämalli

SearchContactViewModel-näkymämalli toteuttaa palvelut, joita hyödynnetään *SearchContactUserControl*-näkymässä. Näkymämalli on esitelty liitteessä 5. Näkymämalli hyödyntää *ContactViewModel*-näkymämallin tarjoamia palveluita. *SearchContactViewModel*-näkymämalli ei ole suoraan yhteydessä sovelluksen malliin vaan tarjoaa rajapinnan sovelluksen malliin *ContactViewModel*-näkymämallin välityksellä. Näkymämallin toiminnallisuudet on kuvattu seuraavassa listassa.

- **public SearchContactViewModel():** Luokan rakentaja. Rakentajassa luodaan lista sovelluksen kaikista yhteyshenkilöistä *ContactViewModel*-näkymämallin *CreateListOfContactViewModel*-metodin avulla.
- **public List<ContactViewModel> ListOfSearchedContacts:** Ominaisuus jonka avulla voidaan asettaa ja hakea lista etsityistä yhteyshenkilöistä. Lista toimii tietomallina *SearchContactUserControl*-näkymän listakomponentille.
- **public ContactViewModel SelectedContact:** Ominaisuus, joka esittää valittua yhteyshenkilöä.
- **public string SearchKey:** Ominaisuus, joka kuvaa merkkijonoa, jonka perusteella järjestelmästä haetaan yhteyshenkilöitä. Merkkijonoa käytetään hakuehtona yhteyshenkilöiden listauksessa.
- **public ICommand SearchCommand:** Komento, jonka avulla päivitetään lista yhteyshenkilöistä. Yhteyshenkilöiden etsintä rajataan *SearchKey*-merkkijonon mukaisesti.

3.3.7 ContactViewModel-näkymämalli

ContactViewModel-näkymämalli tarjoaa rajapinnan, jonka avulla yhden yhteyshenkilön tietoja voidaan muokata ja esittää. Näkymämalli kapseloi sisällensä yhden *Contact*-luokan instanssin, joka edustaa puhelinluettelosovelluksen mallia. Mallin toteutus on kuvattu kappaleessa 3.3.8 Mallin toteutus. *ContactViewModel*-näkymämallin toiminnallisuudet on kuvattu seuraavassa listassa.

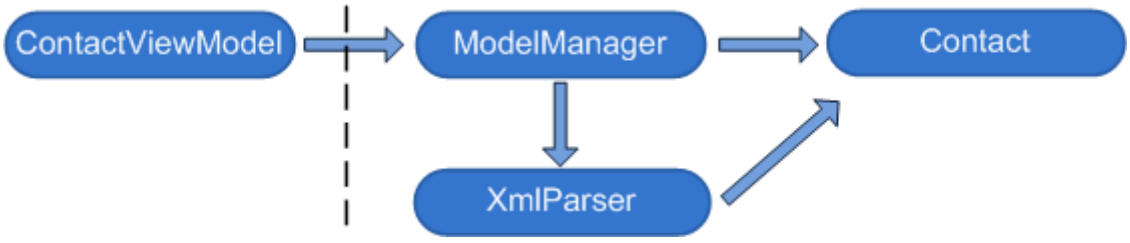
- **public static List<ContactViewModel> CreateListOfContactViewModel (string searchKey):** Staattinen metodi, jonka avulla voidaan luoda lista sovelluksen yhteyshenkilöistä. Yhteyshenkilöt haetaan *searchKey*-merkkijonon mukaisesti. Metodi hyödyntää *ModelManager*-luokan toteutusta yhteyshenkilöiden listaamisessa.
- **public ICommand AddContactCommand:** Komento, jonka avulla yhteyshenkilö voidaan lisätä sovellukseen. Komentoon on toteutettu ehto, joka määrittää milloin komento voidaan suorittaa. Toteutus tarkistaa, onko lisättävän yhteyshenkilön kaikki ominaisuudet asetettu ja ovatko kaikki syötetyt arvot laillisia.
- **string IDataErrorInfo.this[string propertyName]:** Indeksointi, jonka avulla on toteutettu tiedon validointi. Tiedon validointi on toteutettu *Contact*-luokassa. *ContactViewModel*-näkymämalli kutsuu *Contact*-luokassa toteutettua toiminnallisuutta tiedon validoinnissa.

Tämän lisäksi *ContactViewModel*-näkymämalli pitää sisällään ominaisuudet, joiden avulla voidaan muokata ja näyttää *Contact*-luokan ominaisuudet *FirstName*, *Surname*, *Phone*, *Address*, *Postcode* ja *City*. Ominaisuuksien toteutukset ovat liitteessä 6 riveillä 53-149.

Näkymämallin ominaisuuksille on toteutettu tiedon validointi toteuttaen *IDataErrorInfo*-rajapinta. Tiedon validoinnin logiikka on toteutettu *Contact*-luokassa eikä näkymämallissa, joten *ContactViewModel*-näkymämalli ohjaa *IDataErrorInfo*-rajapinnan mukaisen toteutuksen edelleen *Contact*-luokalle. Toteutus on esitelty liitteessä 6 riveillä 179-195.

3.3.8 Mallin toteutus

Puhelinluettelosovelluksen malli muodostuu kolmesta luokasta *Contact*, *ModelManager* ja *XmlParser*. Luokat ovat esitelty liitteissä 7, 8 ja 9. Kuvassa 19 on esitelty puhelinluettelosovelluksen mallin rakenne.



Kuva 19. *Mallin rakenne puhelinluettelosovelluksessa.*

XmlParser-luokka on apuluokka, jonka avulla yhteyshenkilöiden tiedot voidaan kirjoittaa ja lukea xml-tiedostosta. *ModelManager*-luokan avulla voidaan listata, lisätä ja tallentaa sovelluksen yhteyshenkilöitä. Luokka toteuttaa myös yhteyshenkilöiden etsin-

nän annetun merkkijonon perusteella. *Contact*-luokka on sovelluksen tietomalli, joka kuvaa yhden yhteyshenkilön tietoja. *Contact*-luokassa on toteutettu sovelluslogiikka, jonka avulla voidaan validoida syötetyn tiedon laillisuus. Tiedon validointi on toteutettu liitteessä 7 riveillä 67-199.

4 MVVM-SUUNNITTELUMALLIN SOVELTAMINEN OHJELMISTOPROJEKTISSA

Tässä luvussa esitellään toteutusratkaisuja joita on tehty sovellettaessa MVVM-suunnittelumallia Atostekin projektissa. Projektissa toteutettua järjestelmää ja sen arkkitehtuuria käsitellään MVVM-suunnittelumallin kannalta. Luvussa kuvataan, kuinka tyypillisiä käyttöliittymäsovelluksien toiminnallisuuksia toteutetaan MVVM-suunnittelumallin mukaisesti. Käsiteltäviä toiminnallisuuksia ovat näkymien päivittäminen, näkymien sulkeminen, tiedon esittäminen, virheilmoituksien esittäminen, etenemispalkin esittäminen ja mallin toteutus. Tässä luvussa esitellään myös projektissa kehitetty menetelmä näkymämallien yhtenäiseen luomiseen *ViewModelFactory*-luokan avulla.

4.1 Toteutettava järjestelmä

Projektissa toteutettu järjestelmä oli Atostekin asiakkaalle toimitettava asiantuntijatyökalu. Järjestelmä toteutettiin *Windows*-käyttöjärjestelmälle. Toteutettu järjestelmä on mittaussovellus, jonka tarkoitus on testata asiakkaan kehittämän laitteen toimintaa. Järjestelmän toiminta perustuu komentojen lähettämiseen testattavalle laitteelle ja laitteen palauttamien vastauksien ja käyttäytymisen analysointiin. Laitteiden ohjaus ja laitteilta saatavan tiedon käsittely on tärkeässä asemassa toteutetussa järjestelmässä.

Toteutettu järjestelmä on osa suurempaa kokonaisuutta. Asiakkaalla on käytössä useita mittaussovelluksia, jotka testaavat heidän kehittämästä laitteesta eri asioita tai kokonaan eri osia. Korkean tason kuvaus toteutetuista mittauksista tallennetaan kaikille mittaussovelluksille yhteiseen tietokantaan. Varsinainen mittauksen sisältö ja tarkemmat tulokset tallennetaan mittaussovelluksien omaan sovelluskohtaiseen tietokantaan.

Projektissa toteutettu järjestelmä toteutti ensimmäisen vaiheen järjestelmän toiminnasta. Projektin alussa oli jo tiedossa, että toteutettavaan järjestelmään tullaan tulevaisuudessa toteuttamaan uusia lisäominaisuuksia. Tulevaisuudessa toteutettavat lisäominaisuudet tuli ottaa huomioon suunniteltaessa järjestelmän tietokantaa sekä muuta arkkitehtuuria. Arkkitehtuurin tuli mahdollistaa uusien lisäominaisuuksien lisäämisen niin, että vanhoja jo valmiiksi toteutettuja toiminnallisuuksia voitaisiin hyödyntää mahdollisimman paljon. Ensimmäisessä vaiheessa toteutettiin testattavan laitteen toiminnallisuuteen liittyviä testejä. Jatkossa toteutettava lisäosa mahdollistaa testattavan laitteen virrankulutuksen testaamisen.

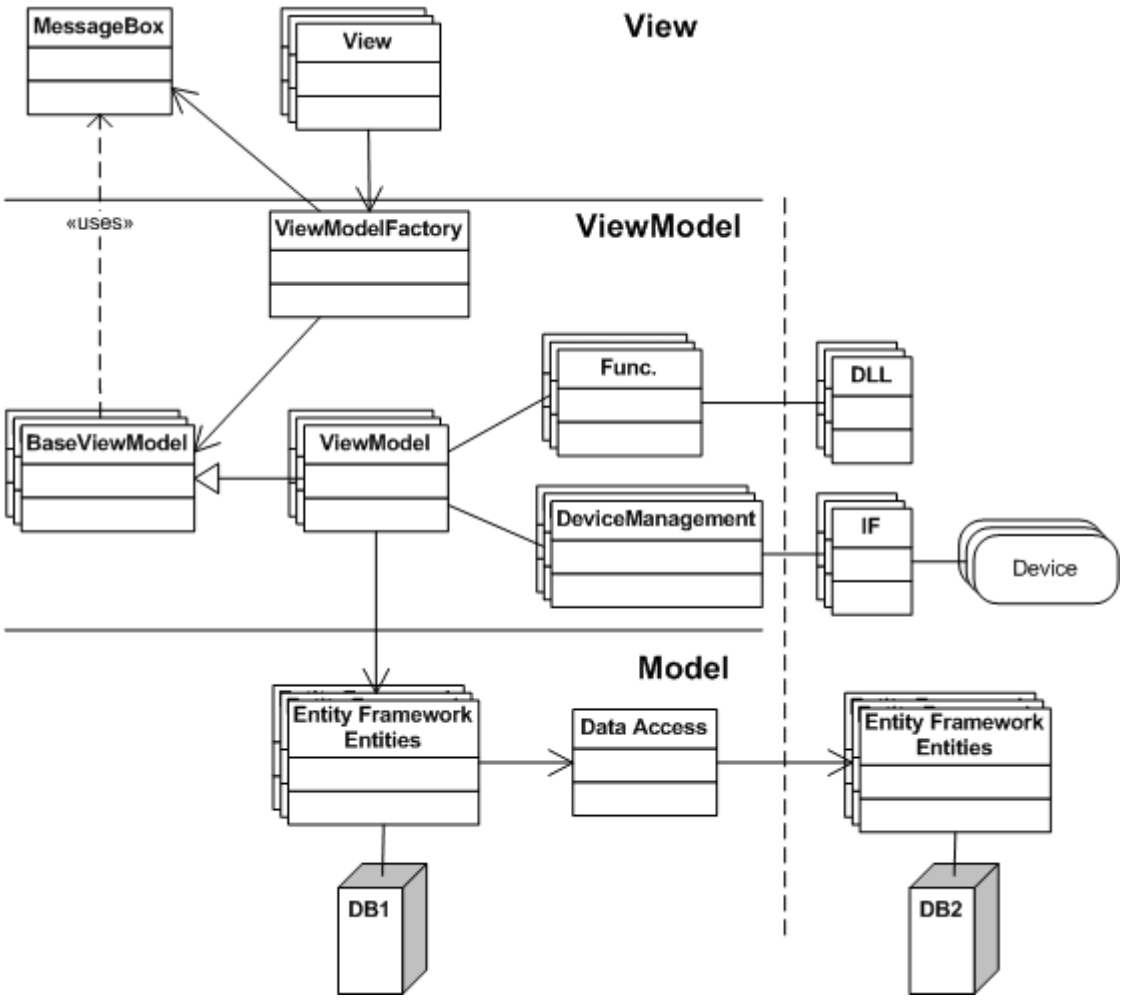
Toteutetun järjestelmän tavoitteena on automatisoida testauksen rutinoituneet testit, jolloin testauksen suorittajille jää enemmän aikaa vaikeasti automatisoitavien testien

suorittamiseen. Järjestelmän arkkitehtuurin on mahdollistettava uusien osien lisääminen järjestelmään ja yleiskäyttöisten komponenttien hyödyntäminen tulevaisuudessa. Projektissa toteutetun järjestelmän vaatimuksena on olla yhteensopiva kaikille mittaussovelluksille yhteisen tietokannan kanssa ja toteuttaa oma sovelluskohtainen tietokanta varsinaisille mittaustuloksille.

Projektissa toteutettu järjestelmä oli Atostekin ensimmäinen WPF-teknologialla toteutettu asiakasprojekti. WPF-teknologian valintaan päädyttiin kolmesta syystä. Ensimmäinen syy oli, että WPF-käyttöliittymäkirjasto on Microsoftin uusin käyttöliittymäkirjasto ja, että Microsoft tukee enemmän WPF-käyttöliittymäkirjaston kehitystä kuin vanhemman *Windows Forms*-käyttöliittymäkirjaston kehitystä. Toinen syy oli, että Atostek tahtoi kokemusta ja osaamista liittyen uuteen WPF-teknologiaan. Kolmantena ja tärkeimpänä syynä WPF-teknologian valintaan oli, että lyhyen selvitystyön jälkeen uskottiin, että WPF-teknologia ja MVVM-suunnittelumalli tarjoaisi hyvän ja tehokkaan tavan toteuttaa asiakkaan vaatima mittausjärjestelmä. WPF-teknologian ja MVVM-suunnittelumallin uskottiin tarjoavan hyvät työkalut modulaarisen, opittavan, testattavan ja ylläpidettävän järjestelmän toteutukselle.

4.2 Järjestelmän arkkitehtuuri

Toteutetun järjestelmän arkkitehtuuri on esitelty kuvassa 20. Arkkitehtuurikuvassa on havainnollistettu toteutetun järjestelmän rakennetta liittyen MVVM-suunnittelumalliin. Arkkitehtuurikuvan tarkoitus ei ole kuvata, mistä luokista järjestelmä koostuu. Kuvan 20 tarkoitus on havainnollistaa järjestelmän rakennetta ja auttaa ymmärtämään MVVM-suunnittelumallin hyödyntämistä järjestelmän arkkitehtuurina.



Kuva 20. *Toteutetun järjestelmän arkkitehtuuri.*

Kuvassa 20 katkoviivan vasemmalla puolella on järjestelmän osat, jotka toteutettiin projektissa. Katkoviivan oikealla puolella on järjestelmän osia, jotka olivat valmiiksi toteutettuja. Valmiiksi toteutettuja osia olivat dll-kirjastot, järjestelmän ohjaamat laitteet ja laitteiden rajapinnat sekä asiakkaan mittaussovelluksille toteutettu yhteinen *DB2*-tietokanta.

Toteutettu järjestelmä on yhteydessä kahteen eri tietokantaan. *DB2*-tietokanta on valmiiksi toteutettu tietokanta, joka edustaa kaikille mittaussovelluksille yhteistä tietoa. Toteutettuun järjestelmään rakennettiin oma järjestelmäkohtainen tietokanta, joka on

yhteensopiva valmiiksi toteutetun tietokannan kanssa. Tiedon haku ja muutokset *DB2*-tietokantaan toteutettiin *Data Access*-komponentin välityksellä. Toteutetun järjestelmän tietokantateknologiana oli *Microsoft SQL Server 2008*. Tietokannan mallintamiseen käytettiin *Entity Framework*-oliomallennusta (Lerman 2010). Mallin toteutus hyödyntäen *Entity Framework*-oliomallennusta on esitelty kappaleessa 4.9 Mallin toteuttaminen.

MVVM-suunnittelumallin mukaisen rakenteen näkymämalli, pitää sisällään toteutetut näkymämallit, mutta myös sovelluslogiikkaa toteuttavat komponentit. Sovelluslogiikka toteuttavat komponentit ovat kuvattu kuvassa 20 *Func*-komponenteilla. Toteutetut näkymämallit pyrittiin pitämään mahdollisimman yksinkertaisina. Näkymämallin tehtävänä on tarjota näkymälle sen tarvitsema toteutus ja tieto. Varsinainen toiminnan sovelluslogiikka toteutettiin *Func*-komponenteissa eikä näkymämalleissa. Näkymämallien toteutus pyrittiin pitämään yksinkertaisena. Loogisena periaatteena oli, että näkymämallit ovat vain käyttöliittymäsovelluksen rungon toteutusta. Varsinainen sovelluslogiikka on toteutettuna *Func*-komponenteissa. Näkymämallit toimivat vain tiedon esittäjänä. Toiminnot, jotka vaativat sovelluslogiikkaa ohjattiin näkymämallin läpi toiminnon suorittavalle komponentille.

Järjestelmän näkymät toteutettiin MVVM-suunnittelumallin mukaisesti. Toteutetut näkymämallit eivät ole tietoisia siitä mitkä järjestelmän näkymät tai muut osat näkymämallien toteutusta käyttävät. Näkymien toteutuksessa poikkeuksena on virheilmoituksien ja etenemispalkin esittäminen. Nämä toiminnallisuudet toteutettiin niin, että näkymämallit hallitsevat dialogien näyttämisen ja sulkemisen. Perustelut suunnitteluratkaisulle ja toiminnallisuuksien toteutukset ovat esiteltynä kappaleissa 4.5 Virheilmoitukset ja 4.6 Toiminnon suorittaminen ja etenemispalkki.

4.3 Näkymämallien luominen

Atostekin projektin aikana kehitettiin oma menetelmä näkymämallien luomiseen. Näkymämallien luominen kapseloitiin *ViewModelFactory*-luokan vastuulle. Menetelmän ansiosta näkymämallien luominen voitiin yhdenmukaistaa koko järjestelmässä, ja näin saatiin selkeä raja näkymien ja näkymämallien välille. *ViewModelFactory*-luokan tehtävänä on tarjota rajapinta kaikkien järjestelmän tuntevien näkymämallien luomiselle, ja pitää kirjaa mitä näkymämalleja järjestelmään on luotu. Luokan avulla kaikki järjestelmän näkymämallit voidaan päivittää kootusti, jolloin tietokannassa tapahtuvat muutokset voidaan päivittää järjestelmän näkymille.

ViewModelFactory -luokan toteutus on esitelty liitteessä 11. *ViewModelFactory*-luokka toteuttaa esimerkin 21 mukaisen metodin, jonka avulla palautetaan instanssi halutusta näkymämallista.

```
public class ViewModelFactory
{
   ...
   private Dictionary<Type, BaseViewModel> _viewModelRepository =
      new Dictionary<Type, BaseViewModel>();

   public BaseViewModel GetViewModelByType(Type vmType)
   {
      if (!_viewModelRepository.ContainsKey(vmType))
      {
         object[] vmArgs = new object[] { _messageBox };
         _viewModelRepository.Add(vmType,
            (BaseViewModel)Activator.CreateInstance(vmType, vmArgs));
      }
      return _viewModelRepository[vmType];
   }
}
```

Esimerkki 21. *ViewModelFactory-luokan toteuttama GetViewModelByType-metodi.*

Esimerkissä 21 luodut näkymämallit tallennetaan *dictionary*-tietorakenteeseen. Tietorakenteen avulla *ViewModelFactory*-luokka palauttaa aina saman instanssin samantyyppisestä näkymämallista. Menetelmän ansiosta näkymän tarvitsema näkymämalli luodaan silloin, kun näkymä avataan ensimmäisen kerran. Tämän jälkeen näkymä voidaan sulkea ja avata uudelleen, jolloin *ViewModelFactory*-luokka palauttaa saman instanssin näkymän käyttämästä näkymämallista. *ViewModelFactory*-luokalta voidaan pyytää haluttu näkymämalli esimerkin 22 mukaisesti.

```
public partial class MainWindow : Window
{
   public MainWindow()
   {
      InitializeComponent();
      this.DataContext =ViewModelFactory.Instance().GetViewModelByType(
                        typeof(ExampleViewModel));
   }
}
```

Esimerkki 22. *Näkymämallin pyytäminen ViewModelFactory-luokan avulla.*

Esimerkissä 21 *MainWindow*-näkymän rakentajassa .cs-kooditiedostossa kutsutaan *ViewModelFactory*-luokan instanssia ja pyydetään *ExampleViewModel*-näkymämallin instanssi.

ViewModelFactory-luokan rakentajassa liitteessä 11 riveillä 12-22 alustetaan *_messageBox*-muuttuja, jota käytetään virheilmoituksien esittämiseen näkymämalleissa. Virheilmoituksien esittämiseen liittyvä toteutus on esitelty kappaleessa 4.5 Virheilmoitukset.

Kaikkien luotujen näkymämallien kapselointi yhden luokan vastuulle mahdollistaa järjestelmän kaikkien näkymämallien päivittämisen. Näkymämallien päivittäminen *ViewModelFactory*-luokassa on toteutettu esimerkissä 23 esitetyn toteutuksen mukaisesti.

```
public class ViewModelFactory
{
   ...

   public void UpdateAllViewModels()
   {
      foreach (var baseViewModelInstance in _viewModelRepository.Values)
      {
         baseViewModelInstance.UpdateViewModelCommand.Execute(null);
      }
   }
}
```

Esimerkki 23. *Näkymämallien päivittäminen ViewModelFactory-luokassa.*

Esimerkissä 23 hyödynnetään *BaseViewModel*-näkymämallin määrittelemää *UpdateViewModelCommand*-komentoa. *BaseViewModel*-näkymämallin toteutus näkymämallin päivittämiseen on esitelty kappaleessa 4.4.1 Näkymämallin päivittäminen.

4.4 BaseViewModel-näkymämalli

BaseViewModel-luokka on abstrakti luokka, joka toteuttaa kaikille näkymämalleille yhteisiä toiminnallisuuksia. Kaikki sovelluksen näkymämallit ovat periytettyjä *BaseViewModel*-näkymämallista. Liitteessä 12 on esitelty *BaseViewModel*-näkymämallin toiminnallisuuksien toteutukset. *BaseViewModel*-näkymämallin avulla kaikille näkymämalleille voidaan suorittaa näkymämallin päivitys (ks. kappale 4.4.1 Näkymämallin päivittäminen), ikkunan sulkeminen (ks. kappale 4.4.2 Ikkunan sulkeminen näkymämallin avulla) ja virheilmoituksien esittäminen (ks. kappale 4.5 Virheilmoitukset).

4.4.1 Näkymämallin päivittäminen

MVVM-suunnittelumallin mukaisesti näkymän ja näkymämallin välinen tiedon välitys toteutetaan WPF-teknologian tiedon sidonnan avulla. MVVM-suunnittelumallissa näkymämallin muutokset välitetään näkymään *INotifyPropertyChanged*-rajapinnan avulla. *BaseViewModel*-näkymämalli toteuttaa logiikan, jonka avulla kaikki näkymämallin toteuttamat ominaisuudet voidaan päivittää yhdellä komennolla. Esimerkissä 24 on esitelty tiedon päivitys *BaseViewModel*-näkymämallin avulla.

```
public abstract class BaseViewModel : INotifyPropertyChanged
{
   private RelayCommand _updateCommand;
   private List<string> _viewModelProperties;

   protected List<string> ViewModelProperties
   {
      get{ return _viewModelProperties; }
      set{ _viewModelProperties = value; }
   }
```

```
public ICommand UpdateViewModelCommand
{
   get
   {
      if (_updateCommand == null)
      {
         _updateCommand = new RelayCommand(
            param => this.UpdateViewModel()
         );
      }
      return _updateCommand;
   }
}

private void UpdateViewModel()
{
   foreach (var propertyName in _viewModelProperties)
   {
      OnPropertyChanged(propertyName);
   }
}
public event PropertyChangedEventHandler PropertyChanged;

protected virtual void OnPropertyChanged(string propertyName)
{
   PropertyChangedEventHandler handler = this.PropertyChanged;
   if (handler != null)
   {
      var e = new PropertyChangedEventArgs(propertyName);
      handler(this, e);
   }
}
```

Esimerkki 24. *Tiedon päivitys BaseViewModel-näkymämallin avulla.*

Esimerkissä 24 *BaseViewModel*-näkymämalli toteuttaa *UpdateViewModelCommand*-komennon. Kun komento suoritetaan, kutsutaan *UpdateViewModel()*-metodia. Metodi käy läpi kaikkien ominaisuuksien nimet, jotka on asetettu *ViewModelProperties*-muuttujaan ja kutsuu *OnPropertyChanged*-tapahtumaa kyseisen ominaisuuden nimellä. Toteutuksen ansiosta kaikki ominaisuudet päivitetään, jotka on asetettu *ViewModelProperties*-muuttujaan.

ViewModelProperties-muuttujan sisältö asetetaan näkymämallissa, joka on periytetty *BaseViewModel*-näkymämallista. Päivitettävien ominaisuuksien asettaminen *ViewModelProperties*-muuttujaan on esitelty esimerkissä 25.

```
public class ExampleViewModel : BaseViewModel
{
   public ExampleViewModel(Func<string, string,
         System.Windows.MessageBoxButton,
         System.Windows.MessageBoxImage,
         System.Windows.MessageBoxResult,
         System.Windows.MessageBoxResult> messageBox)
         : base("ExampleViewModel", messageBox)
   {
   ...
      this.ViewModelProperties = new List<string>()
         { "Property1", "Property2"};
   }
```

```
    public string Property1
    {
       get {...}
       set {...}
    }
    public int Property2
    {
       get {...}
       set {...}
    }
    ...
}
```

Esimerkki 25. *Päivitettävien ominaisuuksien asettaminen BaseViewModel-näkymämallille.*

Esimerkissä 25 *ExampleViewModel*-näkymämalli on periytetty *BaseViewModel*-näkymämallista. Näkymämallin rakentajassa asetetaan näkymämallin toteuttamat ominaisuudet *ViewModelProperties*-muuttujaan. Tämän jälkeen näkymämalli toteuttamat ominaisuudet päivittyvät näkymässä, kun näkymämallin *UpdateViewModelCommand*-komentoa kutsutaan.

4.4.2 Ikkunan sulkeminen näkymämallin avulla

BaseViewModel-näkymämalliin on toteutettu toiminnallisuus, jonka avulla näkymämalli voi sulkea siihen sidotun näkymän. Toiminnallisuuden avulla ikkunan sulkemisen yhteyteen voidaan toteuttaa sovelluslogiikkaa. Ikkunan sulkemisen yhteydessä käyttäjältä voidaan esimerkiksi kysyä, halutaanko tallentamattomat muutokset tallentaa ennen ikkunan sulkemista. Esimerkissä 26 on esitelty toiminnallisuuden toteutus *BaseViewModel*-näkymämallissa.

```
public abstract class BaseViewModel : INotifyPropertyChanged, IDisposable
{
...
   public event Action RequestClose;
   private RelayCommand _closeCommand;
   public System.Windows.Input.ICommand CloseCommand
   {
      get
      {
         if (_closeCommand == null)
         {
            _closeCommand = new RelayCommand(
               param => Close() );
         }
         return _closeCommand;
      }
   }
   public virtual void Close()
   {
      if (RequestClose != null)
      {
         RequestClose();
      }
   }
```

Esimerkki 26. *BaseViewModel-näkymämallin toteutus ikkunan sulkemiseen.*

Esimerkissä 26 on määritelty *RequestClose*-delegaatti ja *CloseCommand*-komento. Komennon suoritus aiheuttaa *Close*-metodin suorittamisen. *BaseViewModel*-näkymämallissa toteutettu *Close*-metodi suorittaa *RequestClose*-delegaattiin asetetun metodin.

BaseViewModel-näkymämallin *RequestClose*-delegaattiin asetetaan haluttu toiminnallisuus näkymän kooditiedostossa. Esimerkissä 27 asetetaan *RequestClose*-delegaattiin ikkunan toteuttama Close-metodi.

```
public partial class MainWindow : Window
{
   public MainWindow()
   {
      InitializeComponent();
      ExampleViewModel vm = ViewModelFactory.Instance().GetViewModelByType(
                            typeof(ExampleViewModel));
      vm.RequestClose += this.Close;

      this.DataContext = vm;
   }
}
```

Esimerkki 27. *Näkymämallin luominen ja Close-metodin välittäminen.*

BaseViewModel-näkymämallin *Close*-metodi on määritelty virtuaaliseksi. Virtuaalinen *Close*-metodi voidaan määritellä uudestaan *BaseViewModel*-näkymämallista periytetyssä näkymämallissa esimerkin 28 mukaisesti.

```
public class ExampleViewModel : BaseViewModel
{
   ...
   public override void Close()
   {
      //Implement logic here...
     base.Close();
   }
}
```

Esimerkki 28. *Close-metodin uudelleen määrittäminen näkymämallissa.*

Esimerkin 28 *Close*-metodissa voidaan toteuttaa haluttu sovelluslogiikka liittyen ikkunan sulkemiseen. Mikäli ikkunaa ei haluta sulkea, pitää metodista poistua ennen kuin kantaluokan *Close*-metodia kutsutaan. Toiminnallisuuden avulla *BaseViewModel*-näkymämallista periytetyssä näkymämallissa voidaan sulkea näkymämalliin sidottu näkymä.

4.5 Virheilmoitukset

Virheilmoitukset ovat tärkeässä asemassa käyttöliittymäsovelluksien toteutuksessa. Virheilmoituksien avulla käyttäjälle voidaan tiedottaa ohjelmassa tapahtuvista virheistä ja laittomista toiminnoista. Virheilmoitusmekanismin avulla käyttäjälle voidaan välittää myös tieto jonkin pitkäkestoisen toiminnon valmistumisesta. Tässä kappaleessa on esitelty Atostekin projektin aikana kehitetty menetelmä virheilmoituksien esittämiseen MVVM-suunnittelumallin mukaisesti.

.Net-käyttöliittymäkirjaston *MessageBox*-luokka tarjoaa valmiin toteutuksen virheilmoituksien esittämiseen (MessageBox 2011). MVVM-suunnittelumallin periaatteisiin kuuluu, että tiedon esittäminen ja näkymien käsittely toteutetaan näkymissä. Tämän ohjeistuksen mukaisesti virheilmoituksien esittäminen tulisi toteuttaa näkymässä. Toisaalta virheilmoituksien esittäminen ja käyttäjän vastauksen tulkitseminen vaativat sovelluslogiikkaa. Sovelluslogiikan ja virheilmoituksien toteuttaminen näkymään ei ole suositeltavaa, koska siinä tapauksessa virheilmoitus olisi käytettävissä vain yhdessä näkymässä. Mikäli jokin toinen näkymä tai komponentti hyödyntäisi samaa toiminnallisuutta, jossa virheilmoitus tulee esittää, pitäisi virheilmoituksen esittäminen toteuttaa uudestaan.

MVVM-suunnittelumallin mukainen virheilmoituksien esittäminen voidaan toteuttaa *ViewModelFactory*-luokan ja *BaseViewModel*-näkymämallin avulla. *BaseViewModel*-näkymämalliin toteutetaan esimerkin 29 mukainen rakentaja ja *_messageBox*-muuttuja.

```
public abstract class BaseViewModel : INotifyPropertyChanged
{
    protected Func<string, string, MessageBoxButton, MessageBoxImage,
    MessageBoxResult, MessageBoxResult> _messageBox;

    ...

    protected BaseViewModel(string viewModelName,
        Func<string, string, MessageBoxButton, MessageBoxImage,
        MessageBoxResult, MessageBoxResult> messageBox)
    {
        _messageBox = messageBox;
        ...
    }
}
```

Esimerkki 29. *Virheilmoitusdelegaatin määrittäminen BaseViewModel-näkymämallissa.*

Rakentaja ottaa vastaan toisena parametrina delegaatin (Func Delegate 2011). Kun näkymämalli luodaan *ViewModelFactory*-luokassa, rakentajan delegaattiin sidotaan .Net-kirjaston *MessageBox*-luokan *Show*-metodi. Esimerkissä 30 on esitetty *BaseViewModel*-näkymämallin luominen ja *MessageBox*-luokan toiminnallisuuden välittäminen näkymämallille.

```
...

messageBox =  (Func<string, string, MessageBoxButton, MessageBoxImage,
   MessageBoxResult, MessageBoxResult>)((msg, capt, buttons, icon, defResult)
   => MessageBox.Show(msg, capt, buttons, icon, defResult));

BaseViewModel baseViewModel = new BaseViewModel("BaseViewModel", messageBox);

...
```

Esimerkki 30. *Virheilmoitusdelegaatin määrittäminen ja BaseViewModel-näkymämallin luominen.*

Tämän jälkeen kaikissa näkymämalleissa, jotka ovat periytyneet *BaseViewModel*-luokasta, voidaan haluttu virheilmoitus näyttää käyttäjälle. Virheilmoituksia ja muita ilmoituksia käyttäjälle voidaan toteuttaa esimerkiksi MVVM-suunnittelumallin mukaisissa komennoissa. Esimerkissä 31 on toteutettu ikkunan sulkemiseen liittyvä ilmoitusdialogi.

```
switch (_messageBox("Do you want to save changes?", "Save changes?",
                    System.Windows.MessageBoxButton.YesNoCancel,
                    System.Windows.MessageBoxImage.Question,
                    System.Windows.MessageBoxResult.Cancel))
{
   case System.Windows.MessageBoxResult.Cancel:

      ...
      break;
   case System.Windows.MessageBoxResult.No:

      ...
      break;
   case System.Windows.MessageBoxResult.Yes:

      ...
      break;
   default:
      ...
      break;
 }
```

Esimerkki 31. *Ilmoitusdialogin toteuttaminen näkymämallissa.*

Esimerkissä 31 *_messageBox()*-metodin kutsuminen aiheuttaa *MessageBox*-luokan *Show()*-metodin kutsumisen. Toteutus luo uuden dialogin, jossa ilmoituksena on ”Do you want to save changes?” ja otsikkona ”Save changes?”. Käyttäjälle annetaan vastausvaihtoehdoiksi ”Yes”, ”No” ja ”Cancel”. Lisäksi dialogissa esitetään *Questinon*-tyyppinen ikoni ja dialogin oletus vastaus on ”Cancel”. Esimerkin 31 mukainen logiikka voidaan toteuttaa esimerkiksi komentoon, joka sulkee ikkunan. Koska dialogin esittäminen on toteutettu näkymämallissa, käyttäjän vastaukseen liittyvä sovelluslogiikka on helppo toteuttaa. Toteutuksen ansiosta aina, kun näkymämallin komento suoritetaan, käyttäjältä pyydetään ohjeistusta, mitä tallentamattomille tiedoille tehdään.

Virheilmoitukset ja dialogit, jotka pyytävät käyttäjältä lisäohjeita toiminnallisuuden suorittamiseen, liittyvät vahvasti järjestelmän sovelluslogiikkaan. Tästä syystä näkymämalli on oikea paikka virheilmoituksien esittämiseen. Esitetyn mekanismin avulla kaikilla sovelluksen näkymämalleilla on työkalunaan *_messageBox*-delegaatti. Delegaatin avulla näkymämallissa on helppo toteuttaa toiminnallisuus, joka pyytää käyttäjältä lisätietoja jonkin operaation suorittamiselle. Varsinaisen dialogin ja sen sisällön esittämisen käyttöliittymässä hoitaa .Net-käyttöliittymäkirjaston *MessageBox*-luokka. *MessageBox*-luokka on kauan käytössä ollut ja hyväksi todettu menetelmä virheilmoituksien esittämiseen käyttäjälle.

4.6 Toiminnon suorittaminen ja etenemispalkki

MVVM-suunnittelumallin mukaisesti, toimintojen suoritus toteutetaan WPF-teknologian komentojen avulla. Pitkäkestoisien toimintojen suorittamiseen vaaditaan kuitenkin enemmän toiminnallisuutta. *BackgroundWorker*-luokka on .Net-kirjaston tarjoama mekanismi, jonka avulla pitkäkestoinen toiminnallisuus voidaan suorittaa omassa säikeessä (BackgroundWorker 2011). Mekanismin ansiosta käyttöliittymäsovelluksen käyttöliittymäsäie vapautetaan toiminnallisuuden suorittamisesta, jolloin käyttöliittymäsäie voi keskittyä käyttöliittymän päivittämiseen. *BackgroundWorker*-luokan toteutus MVVM-suunnittelumallin mukaisesti ei eroa mekanismin normaalista toteuttamisesta. *BackgroundWorker*-luokan ominaisuudet voidaan määritellä ja laukaista normaalisti näkymämallin komennossa. *BackgroundWorker*-luokan toteutuksesta voi lukea lisää *MSDN Library*-dokumentaatioista (BackgroundWorker 2011).

Käyttöliittymäsovelluksissa on yleistä näyttää pitkäkestoisen toiminnallisuuden eteneminen käyttöliittymässä. Projektissa toteutetussa järjestelmässä muun muassa laitteiden yhdistämisessä hyödynnettiin etenemispalkkeja. Tässä kappaleessa esitetään projektissa toteutettu tapa esittää etenemispalkki MVVM-suunnittelumallin mukaisesti.

Etenemispalkin esittäminen MVVM-suunnittelumallin mukaisesti on samantapainen toiminnallisuus kuin virheilmoituksien esittäminen. Näkymien hallinta ja tiedon esittäminen MVVM-suunnittelumallissa kirjallisuuden mukaan kuuluisi näkymien vastuulle, mutta etenemispalkin hallinta vaatii sovelluslogiikkaa ja toiminnallisuutta, joka on järkevää toteuttaa näkymämallissa. Tästä johtuen etenemispalkin esittäminen on toteutettu niin, että näkymämalli pitää sisällään määritellyn ikkunan ja esittää sen käyttäjälle pitkäkestoista toimintoa suoritettaessa.

Etenemispalkkia näyttävä dialogi voidaan toteuttaa yleiskäyttöiseksi koko sovelluksen käytettäväksi. Yleiskäyttöisen etenemispalkin ikkuna on määritelty liitteessä 13. Ikkunan XAML-kuvauksessa, ja .cs-kooditiedostossa esitellään ominaisuuksia joiden avulla etenemispalkki voidaan määritellä ja sen tietoa päivittää.

Esimerkissä 32 on esitelty esimerkkitoteutus, jossa liitteessä 13 esitelty ikkuna alustetaan. Alustus toteutetaan näkymämallin komennossa, joka suorittaa jonkin pitkäkestoisen toiminnallisuuden.

```
_progressWindow = new ProgressWindow();
_progressWindow.TitleText = "Title of the progress dialog...";
_progressWindow.progressBar.IsIndeterminate = false;
_progressWindow.WindowStartupLocation = WindowStartupLocation.CenterScreen;
_progressWindow.DataContext = this;

this.ProgressPercentageValue = 0;
this.ProgressStatus = "Status of the progress...";

...

_progressWindow.ShowDialog();
```

Esimerkki 32. *ProgressWindow-luokan alustus näkymämallin komennossa.*

Esimerkissä 32 alustetaan instanssi *ProgressWindow*-luokasta. Luokan alustuksen jälkeen tulisi *BackgroundWorker*-luokan alustus jossa määritellään suoritettava funktio ja laukaistaan suoritettava toiminto *RunWorkerAsync()*-metodilla. Tämän jälkeen kutsutaan *ProgressWindow*-luokan *ShowDialog()*-metodia, joka aukaisee käyttöliittymään etenemispalkin sisältämän ikkunan. Esimerkissä 32 asetetaan *ProgressWindow*-näkymän *DataContext*-määreeksi se näkymämalli, jossa ikkunan alustus on toteutettu. Tästä seuraa, että liitteessä 13 määritellyssä ikkunan XAML-kuvauksessa käytetyt tiedonsidonta ominaisuudet kohdistuvat tähän samaan näkymämalliin. Näkymämallin on siis toteutettava esimerkissä 33 määritetyt ominaisuudet, joihin ikkunassa on sitouduttu. Tiedon sidonnan avulla näkymä päivittää omaa tietoaan näkymämallin perusteella.

```
public ICommand CancelProgressCommand
{
   get
   {
      if (_cancelWorkCommand == null)
      {
         _cancelWorkCommand = new RelayCommand(
            param => this.CancelWork(),
            param => this.CanCancel
         );
      }
      return _cancelWorkCommand;
   }
}

private void CancelWork()
{
   ...
}
bool CanCancel
{
   ...
}
```

```
public int ProgressPercentageValue
{
   get { return _progressPercentageValue; }
   set
   {
      _progressPercentageValue = value;
      OnPropertyChanged("ProgressPercentageValue");
   }
}

public string ProgressStatus
{
   get { return _progressStatus; }
   set
   {
      _progressStatus = value;
      OnPropertyChanged("ProgressStatus");
   }
}
```

Esimerkki 33. *Liitteessä 13 kuvatun etenemispalkin hyödyntämät ominaisuudet.*

Esimerkin 33 mukaisesti etenemispalkin eteneminen päivitetään *ProgressPercentageValue*-ominaisuuden avulla. *BackgroundWorker*-luokan toteutuksen *ProgressChanged*-tapahtuman käsittelijässä saadaan tieto toiminnon etenemisestä. Tämä tieto voidaan välittää käyttöliittymään asettamalla se *ProgressPercentageValue*-ominaisuuteen.

Ilmoitus pitkäkestoisen toiminnan päättymisestä saadaan *BackgroundWorker*-luokan *RunWorkerCompleted*-tapahtuman välityksellä. Tämän tapahtuman tapahtumankäsittelijässä voidaan käsitellä tulokset, jotka laskenta tuotti ja sulkea dialogi, joka esitti toiminnon suorittamista kutsumalla *ProgressWindow*-luokan *Close()*-metodia.

MVVM-suunnittelumallin mukainen etenemispalkin toteutus hyödyntää hyvin .Net-kirjaston *BackgroundWorker*-luokan toteutusta. Sovelluksen näkymässä ei tarvitse olla tietoinen siitä, kuinka pitkä suoritettava toiminnallisuus tulee olemaan. Näkymässä sitoudutaan kaikkiin toiminnallisuuksiin aina samanlailla WPF-teknologian komentojen avulla. Mikäli suoritettava toiminnallisuus on pitkäkestoinen, näkymämalli toteuttaa sovelluslogiikan, joka suorittaa toiminnon omassa säikeessä ja aukaisee dialogin esittämään toiminnon etenemistä. Tarvittaessa toimintoon voidaan toteuttaa *Cancel*-painike. *Cancel*-toiminnallisuus voidaan ohjata suoraan *BackgrounWorker*-luokan *CancelAsync()*-metodiin, jonka avulla toiminnon suorittaminen voidaan keskeyttää.

WPF-teknologian tiedon sidonta ja tietomallin määrittäminen *DataContext*-määreen avulla mahdollistavat yleiskäyttöisen *ProgressWindow*-ikkunan toteuttamisen. Ikkuna voidaan ottaa käyttöön tässä kappaleessa esitetyllä tavalla useissa eri näkymämalleissa. Ikkunan käyttöönotto ei tarvitse muuta kuin ikkunan alustamisen ja esimerkissä 33 esitettyjen ominaisuuksien toteuttamisen näkymämalliin.

4.7 Tiedon sidonta ja tyyppimuunnokset

WPF-teknologian tiedon sidonta tarjoaa työkaluja tyyppimuunnoksiin tiedon sidonnan yhteydessä. WPF-teknologian *IValueConverter*-rajapinnan ansiosta järjestelmän sama tietomalli voidaan esittää näkymässä usealla eri tavalla. Tässä kappaleessa on esitelty projektissa käytetty menetelmä *byte*-muuttujan esittämiseen kolmella eri tavalla. Projektissa byte-tyyppinen muuttuja esitettiin näkymässä desimaali-, heksadesimaali- ja binäärilukuna.

Liitteessä 14 on esitelty *ByteToHexConverter*- ja *ByteToBinConverter*-luokat. Molemmat luokat toteuttavat *IValueConverter*-rajapinnan. Esimerkissä 34 on esitelty *ByteToHexConverter*-luokan toteutus.

```
[ValueConversion(typeof(byte), typeof(string))]
public class ByteToHexConverter : IValueConverter
{
   object IValueConverter.Convert(object value, Type targetType,
      object parameter, System.Globalization.CultureInfo culture)
   {
      return ((byte)value).ToString("X2");
   }

   object IValueConverter.ConvertBack(object value, Type targetType,
      object parameter, System.Globalization.CultureInfo culture)
   {
      byte retval = 0x00;
      bool valid = Byte.TryParse((string)value,
         System.Globalization.NumberStyles.AllowHexSpecifier, null,
         out retval);

      if (valid)
         return retval;
      else
         return DependencyProperty.UnsetValue;
   }
}
```

Esimerkki 34. *IValueConverter-rajapinnan toteuttama ByteToHexConverter-luokan toteutus.*

Esimerkissä 34 esitetty luokka toteuttaa *Convert*- ja *ConvertBack*-metodit. *Convert*-metodi muuttaa annetun *byte*-muuttujan arvon heksadesimaaliksi, ja *ConvertBack*-metodi muuttaa heksadesimaaliarvon takaisin *byte*-muuttujaksi. Liitteessä 14 esitelty *ByteToBinConverter*-luokka sisältää toteutuksen, jonka avulla *byte*-muuttuja voidaan muuttaa *bitti*-merkkijonoksi ja *bitti*-merkkijono takaisin *byte*-muuttujaksi. Esimerkissä 35 on esitelty näkymä, joka hyödyntää *ByteToHexConverter*- ja *ByteToBinConverter*-luokkien toteutusta.

```
<Window x:Class="Examples.ValueConverters"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:converters="clr-namespace:Examples.ValueConverter"
      Title="ValueConverters" Height="148" Width="184">

    <Window.Resources>
        <converters:ByteToHexConverter x:Key="byteToHexConverter" />
        <converters:ByteToBinConverter x:Key="byteToBinConverter" />
    </Window.Resources>

    <Grid>
        ...
        <Label ... Content="Dec:"/>
        <TextBox ... Text="{Binding Path=ByteValue}"/>

        <Label ... Content="Hex:"/>
        <TextBox ... Text="{Binding Path=ByteValue,
                            Converter={StaticResource byteToHexConverter}}"/>

        <Label ... Content="Bin:"/>
        <TextBox ... Text="{Binding Path=ByteValue,
                            Converter={StaticResource byteToBinConverter}}"/>
    </Grid>
 </Window>
```

Esimerkki 35. *ByteToHexConverter- ja ByteToBinConverter-luokkia hyödyntävä näkymä.*

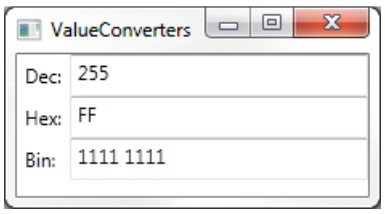
Esimerkin 35 näkymässä on määritelty kolme *TextBox*-komponenttia. Jokaisen *TextBox*-komponentin *Text*-kenttä on sidottu tietomallin *ByteValue*-ominaisuuteen. Liitteessä 14 esitetyt tyyppimuuntimet on lisätty kahden alimmaisen *Text*-kentän tiedon sidontaan *Converter*-määreen avulla. WPF-teknologian tiedon sidonnan ansiosta tyyppimuuntimien *Convert*- ja *ConvertBack*-metodeja kutsutaan aina, kun tietoa välitetään tietomallista näkymään tai näkymästä tietomalliin. Esimerkissä 36 on esitelty tietomallin määrittäminen näkymän .cs-kooditiedostossa. Esimerkin 36 selkeyden vuoksi tietomallina käytetty *DataViewModel*-luokka on esitelty ja luotu näkymän yhteydessä.

```
namespace Examples
{
    public partial class ValueConverters : Window
    {
        public ValueConverters()
        {
            InitializeComponent();
            this.DataContext = new DataViewModel(0xff);
        }
    }

    public class DataViewModel
    {
        private byte _byteValue;
        public DataViewModel(byte byteValue)
        {
            _byteValue = byteValue;
        }
```

```
        public byte ByteValue
        {
            get { return _byteValue; }
            set { _byteValue = value; }
        }
    }
}
```

Esimerkki 36. *Tietomallin määrittäminen ja näkymän alustus.*

Esimerkissä 36 näkymän *DataContext*-määreeseen asetetaan *DataViewModel*-luokan instanssi. *DataViewModel*-luokka toteuttaa *ByteValue*-ominaisuuden, joka palauttaa *byte*-muuttujan. Näkymälle asetettu tietosisältö alustetaan 0xff-arvolla. Kuvassa 21 on esitelty esimerkin 35 muodostama näkymä.

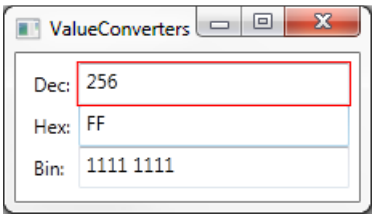


Kuva 21. *Esimerkin 35 muodostama näkymä.*

Kuvan 21 esittämässä näkymässä tietomallin *ByteValue*-ominaisuus esitetään kolmella eri tavalla. Ylimmäisessä *TextBox*-komponentissa *ByteValue*-ominaisuus esitetään desimaalilukuna, keskimmäisessä heksadesimaalina ja alimmaisessa binäärisenä merkkijonona.

Ylimmäisessä desimaalilukuesityksessä ei ole käytetty tyyppimuuntajaa. WPF-teknologian tiedon sidonta käyttää oletusarvoisesti *ToString()*-metodia tiedon esittämiseen *TextBox*-komponentissa. Oletusarvoinen *byte*-muuttujan *ToString()*-metodin palauttaa muuttujan sisällön desimaalilukuna. Oletustoteutus osaa muuttaa myös käyttäjän antaman desimaaliluvun takaisin *byte*-muuttujaksi.

Käyttäjän syöttämälle tiedolle ei ole tässä esimerkissä toteutettu tiedon validointia. WPF-teknologian tiedon sidonta osaa kuitenkin käsitellä näkymään syötetyn laittoman tiedon. Kuvassa 22 on esitelty tilanne jossa käyttäjä on syöttänyt ylimpään *TextBox*-komponenttiin arvon 256. Arvo 256 ylittää *byte*-muuttujan arvoalueen, jolloin muunnosoperaatio synnyttää *OverflowException*-poikkeuksen. WPF-teknologian tiedon sidonta käsittelee poikkeuksen ja ilmoittaa tapahtuneesta virheestä rajaamalla *TextBox*-komponentin punaisella kehyksellä.



Kuva 22. *Virheellinen arvo tyyppimuunnoksessa.*

WPF-teknologian tiedon sidonta ja tyyppimuunnos *IValueConverter*-rajapinnan avulla tarjoavat tehokkaan ja yleiskäyttöisen tavan esittää järjestelmän tietomallia näkymässä usealla eri tavalla.

4.8 Tiedon esittäminen taulukossa

Toteutetussa järjestelmässä tärkeässä asemassa oli testiskriptien luominen. Skriptit muodostuvat komennoista, joita lähetetään laitteelle. Järjestelmässä on määritelty noin sata erilaista komentoa. Tämän lisäksi käyttäjä pystyy luomaan uusia komentoja järjestelmään. Komentojen yksinkertaistettu rakenne koostuu nimestä, parametreista ja komennon tyypistä. Komennolle asetettavat parametrit riippuvat komennon tyypistä. Järjestelmä tunnistaa noin kaksikymmentä erilaista komentotyyppiä. Tässä kappaleessa on esitelty yksinkertaistettu toteutus projektista käytetystä taulukosta, joka mahdollistaa testiskriptien luomisen.

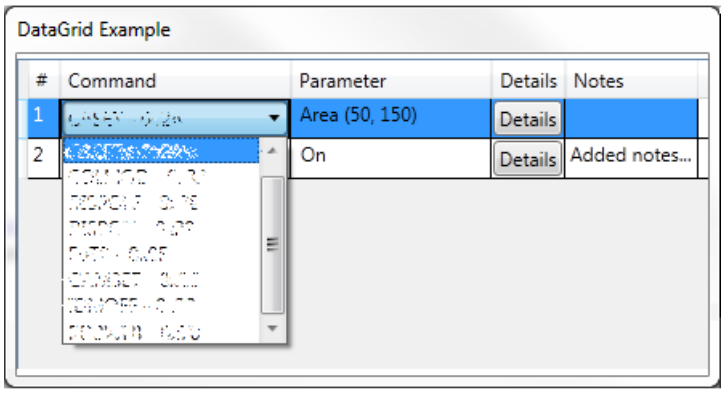
Skriptin luomiseen hyödynnettiin WPF-teknologian *DataGrid*-komponenttia. *DataGrid*-komponentti on taulukko johon voidaan määritellä tarvittavat sarakkeet. Yksi rivi edustaa yhtä laitteelle lähetettävää komentoa. Toteutusteknisesti haastavimpana ominaisuutena oli parametrien esittäminen valitulle komennolle. Koska komento voi sisältää erilaisia parametreja, tuli käyttäjälle tarjota myös erilaisia käyttöliittymäkomponentteja parametrien asettamiseen riippuen valitusta komennosta. Esimerkissä 37 on esitetty yksinkertaistettu *DataGrid*-komponentti, jonka avulla skriptin komentoja voidaan käsitellä.

```
<DataGrid ... ItemsSource="{Binding Path=ScriptSteps}">
   <DataGrid.Columns>
      <DataGridTextColumn Header="#" Binding="{Binding Index}"/>

      <DataGridTemplateColumn Header="Command">
         <DataGridTemplateColumn.CellTemplate>
            <DataTemplate>
               <TextBlock Text="{Binding Command}"/>
            </DataTemplate>
         </DataGridTemplateColumn.CellTemplate>
         <DataGridTemplateColumn.CellEditingTemplate>
            <DataTemplate>
               <ComboBox ItemsSource="{Binding Path=CommandList}"
                         SelectedItem="{Binding Path=Command}"/>
            </DataTemplate>
         </DataGridTemplateColumn.CellEditingTemplate>
      </DataGridTemplateColumn>
```

```
        <DataGridTemplateColumn Header="Parameter"
           CellTemplate="{StaticResource NotSelectedTemplate}"
           CellEditingTemplate="{StaticResource ParameterEditingDataTemplate}" >
        </DataGridTemplateColumn>

        <DataGridTemplateColumn Header="Details">
           <DataGridTemplateColumn.CellTemplate>
              <DataTemplate>
                 <Button Content="Details"
                         Command="{Binding Path=OpenDetailsCommand}" />
              </DataTemplate>
           </DataGridTemplateColumn.CellTemplate>
        </DataGridTemplateColumn>

        <DataGridTextColumn Header="Notes" Binding="{Binding Notes}"/>

     </DataGrid.Columns>
  </DataGrid>
```

Esimerkki 37: *Skriptin esittäminen DataGrid-komponentissa.*

Esimerkissä 37 on esitelty *DataGrid*-komponettilla toteutettu taulukko, jossa on viisi saraketta. Taulukon rivit muodostuvat tietomallin alkioista, jotka asetetaan taulukolle. Taulukolle asetettu tietomalli on *ItemsSource*-määreeseen asetettu *ScriptSteps*-muuttuja. *ScriptSteps*-muuttuja on lista *ScriptStepViewModel*-instansseja. Jokainen *ScriptStepViewModel*-instanssi edustaa yhtä riviä taulukossa. *ScriptStepViewModel*-instanssi toteuttaa ominaisuudet, joita näkymä käyttää tiedon sidonnan avulla. Esimerkin 37 muodostama näkymä on esitelty kuvassa 23. Kuvassa 23 esitettyä listaa järjestelmän komennoista ei voida näyttää sopimusteknisistä syistä.



Kuva 23. *Esimerkin 37 muodostama näkymä.*

Esimerkissä 37 määritellään jokaiselle sarakkeelle esitystapa, jonka mukaan tieto esitetään sarakkeessa. Tiedon esittämiseen käytetään WPF-teknologian tiedon sidontaa. Ensimmäisessä sarakkeessa esitetään *ScriptStepViewModel*-näkymämallin toteuttama *Index*-muuttuja. Muuttuja esitetään *DataGridTextColumn*-komponentissa. Seuraavassa *Command*-sarakkeessa hyödynnetään *CellTemplate* ja *CellEditingTemplate*-esitysmalleja tiedon esittämiseen. *CellTemplate*-esitysmalli määrittelee komponentit joita käytetään sarakkeen tiedon esittämiseen kun käyttäjä ei ole valinnut saraketta. *Cel-*

lEditingTemplate-esitysmallia käytetään silloin kun käyttäjä on valinnut kyseisen solun taulukosta ja muokkaa sitä. Esimerkin 34 mukaisesti *Command*-sarakkeessa esitetään näkymämallin *Command*-muuttuja *TextBlock*-komponentissa, kun solu ei ole valittuna ja sitä ei muokata. Kun käyttäjä valitsee solun ja muokkaa sen tietoa, käyttäjälle näytetään lista valittavista komennoista *ComboBox*-komponentin avulla.

Parameter-sarakkeen määrittäminen on kaikista monimutkaisin. *Parameter*-sarakkeessa ei riitä, että käytettävä esitysmalli vaihdetaan sen mukaan muokkaako käyttäjä solua vai ei. Tämän lisäksi esitysmallin valintaa vaikuttaa tieto siitä minkä tyyppinen komento kyseisellä rivillä on valittuna.

Esimerkin 37 mukaisesti *Parameter*-sarakkeessa käytetään *NotSelectedTemplate*-esitysmallia kun käyttäjä ei muokkaa solua. Esimerkissä 38 on esitelty *NotSelectedTemplate*-esitysmallin määrittely. Esimerkin 38 mukaisesti *NotSelectedTemplate*-esitysmallissa näytetään valitun komennon parametreista tietoa *ParameterInfo*-ominaisuuden avulla.

```
<DataTemplate x:Key="NotSelectedTemplate">
   <Grid>
      <Grid.ColumnDefinitions>
         <ColumnDefinition Width="*"/>
      </Grid.ColumnDefinitions>
      <Label Content="{Binding Path=ParameterInfo}" Margin="0,-5,0,0" />
   </Grid>
</DataTemplate>
```

Esimerkki 38. *NotSelectedTemplate-esitysmallin märittely.*

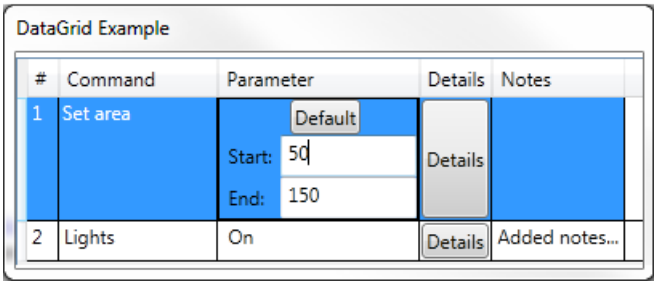
Esimerkin 37 mukaisesti *Parameter*-sarakkeessa käytetään *ParameterEditingDataTemplate*-esitysmallia, kun käyttäjä muokkaa solua. Esimerkissä 39 on esitelty *ParameterEditingDataTemplate*-esitysmallin määrittely.

```
<DataTemplate x:Key="ParameterEditingDataTemplate">
   <Control x:Name="parameterControl"
            Template="{StaticResource ReadOnlyTemplate}" />
   <DataTemplate.Triggers>

      <DataTrigger Binding="{Binding ParameterType}">
         <DataTrigger.Value>
            <commonUtils:ParameterType>OnOff</commonUtils:ParameterType>
         </DataTrigger.Value>
         <Setter TargetName="parameterControl" Property="Template"
                 Value="{StaticResource OnOffTemplate}" />
      </DataTrigger>

      <DataTrigger Binding="{Binding ParameterType}">
         <DataTrigger.Value>
            <commonUtils:ParameterType>StartEnd</commonUtils:ParameterType>
         </DataTrigger.Value>
         <Setter TargetName="parameterControl" Property="Template"
            Value="{StaticResource StartEndTemplate}" />
      </DataTrigger>
   </DataTemplate.Triggers>
</DataTemplate>
```

Esimerkki 39. *ParameterEditingDataTemplate-esitysmallin määrittely.*

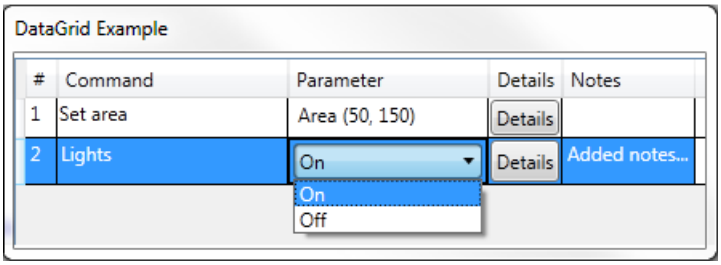
Esimerkissä 39 on havainnollistettu tilannetta jossa järjestelmä tuntee vain kaksi eri parametri tyyppiä. Mikäli riville valitun komennon tyyppi on *OnOff*-tyyppinen, esitetään komennon parametrit *OnOffTemplate*-esitysmallilla. *StartEndTemplate*-esitysmallia käytetään tilanteessa, jossa valitun komennon tyyppi on *StartEnd*-tyyppinen. Esitysmallin vaihtaminen komennon tyypin perusteella on toteutettu WPF-teknologian *Control*-komponentin ja *DataTrigger*-ominaisuuden avulla. Esimerkissä määritellään *parameterControl* niminen *Control*-komponentti, jonka *Template*-määreeseen asetetaan *DataTrigger*-ominaisuuden avulla haluttu esitysmalli. *OnOffTemplate*- ja *StartEndTemplate*-esitysmallit ovat määritelty liitteessä 15 riveillä 26-53.

Kuvassa 24 on esitelty näkymä, jossa käyttäjä on muokkaamassa *StartEnd*-tyyppistä komentoa. Kuvissa 24 ja 25 näkyvät komennot eivät liity toteutettuun järjestelmään.



Kuva 24. *StartEnd-tyyppisen komennon muokkaaminen.*

Kuvassa 25 on esitelty näkymä, jossa käyttäjä on muokkaamassa *OnOff*-tyyppistä komentoa.



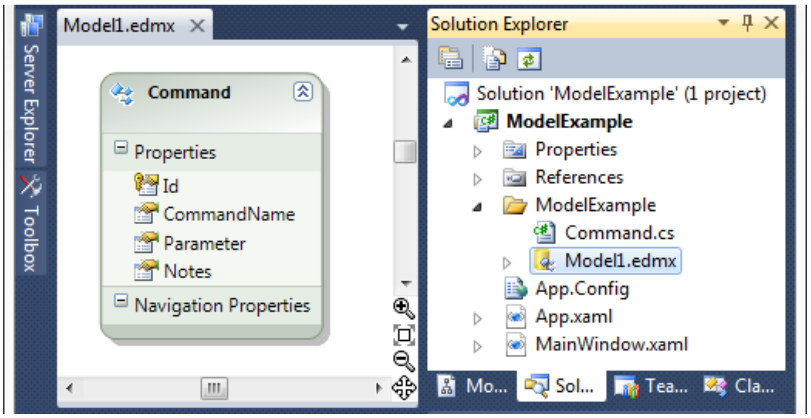
Kuva 25. *OnOff-tyyppisen komennon muokkaaminen.*

Liitteessä 15 on esitelty koko näkymän määritys XAML-kuvauskielellä.

4.9 Mallin toteuttaminen

Projektissa järjestelmän malli toteutettiin hyödyntämällä *EntityFramework*-kehystä. Tässä kappaleessa esitellään yksinkertaisen mallin luominen *Entity Framework*-kehyksen avulla, ja kuvataan, kuinka *Entity Framework*-kehyksellä toteutettua mallia voidaan hyödyntää MVVM-suunnittelumallin mukaisessa järjestelmässä. Lisätietoja *Entity Framework*-kehyksestä ja tarkemmat toteutusohjeet voi lukea Lermanin (2010) kirjasta Programming Entity Framework.

Projektissa toteutetun järjestelmän malli toteutettiin hyödyntäen *Entity Framework*-kehyksen malli ensin -suunnittelua (eng Model-first design). *Visual Studio 2010* on toteutettu *EDMD*-työkalua (eng. *Entity Data Model Designer*), jonka avulla järjestelmän tietokanta ja tietomalli voidaan toteuttaa. *Visual Studio 2010* projektiin voidaan lisätä *ADO.NET Entity Data Model*-elemetti. Elementin avulla voidaan määritellä järjestelmän tietomalli. Kuvassa 26 on esitelty *EDMD*-työkalulla määritelty järjestelmän tietomalli, joka pitää sisällään *Command*-entiteetin.



Kuva 26. *Command-entiteetin määrittäminen EntityFramework-kehyksen avulla.*

Command-entiteetille on määritelty ominaisuudet *Id*, *CommandName*, *Parameter* ja *Notes*. *EntityFramework*-kehyksen avulla määritetystä tietomallista voidaan luoda tietokantaskripti, jonka avulla järjestelmän tietokanta luodaan. Kuvassa 26 määritelty tietomalli on erittäin yksinkertainen, eikä siinä ole esitelty kuin yksi entiteetti. Todellisessa järjestelmässä entiteettejä on useita, ja niiden välille on toteutettu suhteita. *EntityFramework*-kehyksen laajempi toteuttaminen ja käyttäminen eivät kuulu tämän diplomityön aiheeseen. Lisää *EntityFramework*-kehyksestä voi lukea Lermanin (2010) kirjasta.

MVVM-suunnittelumallin kannalta tärkein ominaisuus *EntityFramework*-kehyksen käyttämisessä on entiteettien *partial*-luokkamääritykset. *EntityFramework*-kehys luo oletustoteutukset määritetyille entiteeteille ja ominaisuuksille. Entiteeteille voidaan toteuttaa lisää toiminnallisuutta hyödyntämällä *partial*-luokkamäärettä. Esimerkissä 40 on lisätty Command-entiteettiin uusi toteutus ToString()-metodista.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ModelExample.ModelExample
{
   partial class Command
   {
      public override string ToString()
      {
         return this.CommandName;
      }
   }
}
```

Esimerkki 40. *ToString-metodin uudelleen määrittäminen Command-entiteetissä.*

EntityFramework-kehyksen avulla järjestelmän tietokanta voidaan määritellä entiteetti-luokkina. MVVM-suunnittelumallin mukaisessa järjestelmässä nämä luokat toteuttavat järjestelmän mallin. Entiteetti-luokkiin voidaan toteuttaa lisää sovelluslogiikkaa ja toiminnallisuutta tiedon hallinnalle. Esimerkiksi kappaleessa 3.2.5 sivulla 29 esitelty tiedon validoinnin toteutus *IDataErrorInfo*-rajapinnan avulla voidaan toteuttaa entiteettiluokalle *partial*-luokkamäärityksen avulla.

5 MVVM-SUUNNITTELUMALLIN ARVIOINTI

Projektissa sovellettua MVVM-suunnittelumallia arvioitiin toteutuksen modulaarisuuden, suorituskyvyn, osien uudelleenkäytettävyyden, uuden kokonaisuuden lisäämisen sekä opittavuuden näkökulmista. Tässä luvussa on käsitelty jokaista arviointikohtaa omissa kappaleissaan.

5.1 Modulaarisuus

MVVM-suunnittelumalli jakaa järjestelmän kolmeen suureen kokonaisuuteen. Projektissa saatujen havaintojen perusteella MVVM-suunnittelumallin modulaarisuus toteutuu hyvin myös käytännössä. Järjestelmä jakaantuu selkeästi näkymään, näkymämalliin ja malliin. Jokainen kokonaisuus sisältää useita aliluokkia, jotka toteuttavat toiminnallisuuden kyseiselle kokonaisuudelle.

Arkkitehtuurien soveltamisessa on tärkeää, että arkkitehtuurin rakenne voidaan tunnistaa myös käytännössä. MVVM-suunnittelumallin rakenne on hyvin havaittavissa sivulla 18 esitetyssä kuvassa 9. Kuvasta 9 voidaan nähdä, kuinka suunnittelumallin eri osat jakautuvat myös toteutusympäristössä eri kansioihin ja kansioiden alla oleviin kooditiedostoihin. Kuvassa 9 esitetty rakenne on peräisin esimerkkisovelluksesta, mutta sama rakenne toteutui myös sovellettaessa MVVM-suunnittelumallia ohjelmistoprojektissa. Toteutusympäristössä muodostuvan rakenteen ansiosta järjestelmän rakenne on helposti tunnistettavissa, ja toteutuksen seuraaminen on helpompaa. Käytännössä muodostuva tiedostojen ja kansioiden rakenne ohjelmointiympäristöön on todiste siitä, että MVVM-suunnittelumallin teoriassa esitetty rakenne on toimiva kokonaisuus myös käytännössä toteutettaessa.

Hyvän modulaarisuuden MVVM-suunnittelumallissa mahdollistaa tiedon sidonta. Tiedon sidonnan avulla näkymän ja näkymämallien välinen tiedon välitys ja toimintojen suoritus on helppo toteuttaa. Tiedon sidonta ratkaisee näkymän erottamisen sovelluslogiikasta, joka on malli-näkymä-arkkitehtuurien yleinen tavoite. WPF-teknologian näkymien määrittäminen XAML-kuvauskielillä ja tiedon sidonta mahdollistavat, että näkymät ovat irrallisia muusta järjestelmän logiikasta. Näkymät toimivat puhtaasti näkymämallien kapseloiman sovelluslogiikan tarkkailijoina. Modulaarisuus saavutetaan WPF-teknologian tarjoamilla työkaluilla ja toteutusratkaisuilla, joten yksittäisen toteuttajan ei tarvitse toteuttaa menetelmiä aina uudestaan uuden järjestelmän kohdalla.

5.2 Suorituskyky

MVVM-suunnittelumallissa kriittisin suorituskykyä vaativa osa-alue on näkymän ja näkymämallin välinen kommunikointi tiedon sidonnan avulla (ks. kappale 3.2.4 Tiedon sitominen). WPF-teknologiassa tiedon sidontaan on toteutettu paljon valmiita ominaisuuksia. WPF-teknologian ominaisuuksien ansiosta näkymän ja näkymämallin välinen tiedon sidonta on yksinkertainen toteuttaa. Käytettävät ominaisuudet eivät kuitenkaan anna mahdollisuutta vaikuttaa toteutuksen suorituskykyyn.

Atostekin projektissa toteutettu järjestelmä listaa suoritettuja mittauksia taulukon muodossa. Yksi mittaus koostuu sadoista komennoista, jolloin taulukon rivimäärä on myös satoja. Projektissa tehtyjen havaintojen perusteella WPF-teknologian tiedon sidonta on riittävän suorituskykyinen. WPF-teknologian *DataGrid*-komponentti optimoi ladattavan tiedon määrää lataamalla näytölle vain ne elementit, joita näkymässä näytetään. Tämän ominaisuuden seurauksena näkymässä on havaittavissa lataamista, kun käyttäjä näyttää taulukon uusia rivejä vetämällä listaa alaspäin. Lataaminen on havaittavissa vain, kun lista esitetään ensimmäisen kerran näkymässä. Listan esittäminen nopeutuu, kun käyttäjä on käynyt läpi koko listan.

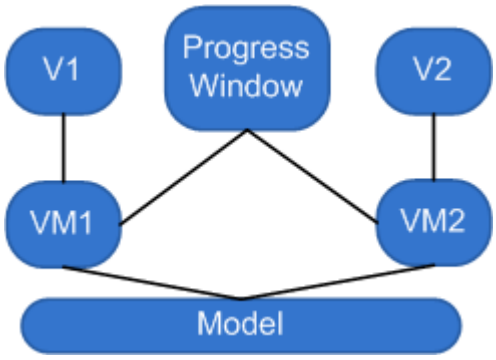
MVVM-suunnittelumallissa toinen kriittinen toteutusyksityiskohta suorituskyvyn kannalta on toiminnon suorittaminen komentojen avulla (ks. kappale 3.2.7 Toiminnon suorittaminen). WPF-teknologian komennoille voidaan toteuttaa *CanExecute*-metodi, joka määrittelee, voidaanko toiminto suorittaa vai ei. WPF-teknologian komentojen käsittelijä kutsuu kaikkien määriteltyjen komentojen *CanExecute*-metodia usein. Mikäli metodien toteutus ei ole yksinkertainen ja nopea, järjestelmän käyttö hidastuu.

5.3 Näkymän uudelleenkäytettävyys

MVVM-suunnittelumalli mahdollistaa saman näkymän uudelleenkäyttämisen. Näkymät on määritelty XAML-kuvauskielellä, eivätkä ne pidä sisällään sovelluslogiikkaa. Näkymät toimivat näkymämallin tai näkymämallien tarkkailijana, jolloin yksi näkymä ei ole sidoksissa johonkin tiettyyn näkymämalliin. Näkymän uudelleenkäytettävyyden mahdollistaa WPF-teknologian tiedon sidonta. Tiedon sidonnan ansiosta näkymissä määritellään pelkästään ominaisuuksien nimiä, joiden avulla tunnistetaan käytettävä tieto tai suoritettava toiminnallisuus.

Projektissa toteutettu *ProgressWindow*-dialogi on esimerkki uudelleenkäytettävästä näkymästä. *ProgressWindow*-dialogi on määritelty liitteessä 13. Liitteen 13 määrittämässä näkymässä esitellään *ProgressPercentageValue*-, *ProgressStatus*- ja *CancelProgressCommand*-ominaisuudet. Vastuu näiden ominaisuuksien toteutuksesta on näkymään sidotulla tietomallilla eli näkymämallilla. Liitteessä 13 määriteltyä näkymää voidaan uudelleenkäyttää järjestelmän eri osissa. Näkymän uudelleenkäyttäminen vaatii, että näkymämallissa on toteutettu näkymän käyttämät ominaisuudet. Kuvassa 27 on havainnollistettu näkymän uudelleenkäytettävyyttä. Kuvassa 27 on esitelty kaksi näkymää (*V1, V2*) ja näiden näkymien näkymämallit (*VM1, VM2*). Molempiin näkymämal-

leihin (*VM1*, *VM2*) on toteutettu *ProgressWindow*-dialogin vaatimat ominaisuudet. Samaa ProgressWindow-dialogin toteutusta voidaan uudelleenkäyttää järjestelmässä, koska molemmat näkymämallit toteuttavat *ProgressWindow*-dialogin vaatimat ominaisuudet.



Kuva 27. *ProgressWindow-dialogin uudelleenkäyttäminen.*

Kuvassa 27 havainnollistetussa esimerkissä uudelleenkäytettävä näkymä oli *Window*-komponentti eli kokonainen ikkuna. Uudelleenkäytettävä näkymä voi olla myös yksi näkymän osa. Samaa menetelmää käyttäen näkymissä voidaan määritellä yleiskäyttöisiä *UserControl*-komponentteja, joissa määritellään, kuinka jokin tieto esitellään näkymässä. Määriteltyjä *UserControl*-komponentteja voidaan käyttää järjestelmän eri näkymissä. Samalla tavalla näkymiin sidotuissa näkymämalleissa pitää toteuttaa *UserControl*-komponentin määrittelemät ominaisuudet.

5.4 Näkymämallin uudelleenkäytettävyys

MVVM-suunnittelumallissa näkymämallit voidaan toteuttaa uudelleenkäytettäviksi. Uudelleenkäytettävät näkymämallit tarjoavat toiminnallisuuden ja tiedon jostain järjestelmän kokonaisuudesta. Uudelleenkäytettävä näkymämalli tarkoittaa käytännössä sitä, että järjestelmän yksi näkymä tarjoaa käyttöliittymän tiedon muokkaamiselle. Tietoa muokkaava näkymä hyödyntää uudelleenkäytettävää näkymämallia tiedon lisäämiseen, päivittämiseen ja hallintaan. Järjestelmän toinen näkymä tarjoaa käyttöliittymän saman tiedon esittämiselle. Näkymä, jossa tietoa esitetään hyödyntää samaa näkymämallia, jolloin samaan tietoon liittyvää toiminnallisuutta ei tarvitse toteuttaa useaan kertaan, vaan voidaan käyttää uudelleenkäytettävää näkymämallia.

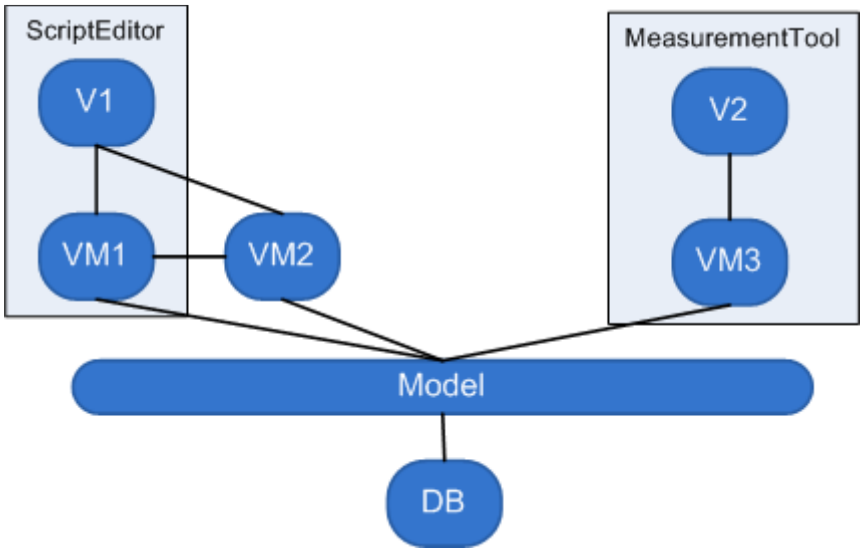
Esimerkki uudelleenkäytettävästä näkymämallista on *ContactViewModel*-näkymämalli, jonka rakenne ja käyttö esiteltiin kappaleessa 3.3 Puhelinluettelosovelluksen toteutus MVVM-suunnittelumallilla. *ContactViewModel*-näkymämalli toteuttaa toiminnallisuuden yhteyshenkilön käsittelyyn ja tarjoaa ominaisuudet yhteyshenkilön tietojen hallintaan. Samaa näkymämallia hyödynnetään *AddContactUserControl*- ja *SearchContactUserControl*-näkymissä. *AddContactUserControl*-näkymässä *ContactViewModel*-näkymämallia käytetään yhteyshenkilön lisäämiseen ja sen tietojen muok-

kaamiseen. *SearchContactUserControl*-näkymässä saman näkymämallin samoja ominaisuuksia käytetään yhteyshenkilön tietojen esittämiseen. Esimerkki uudelleenkäytettävästä näkymämallista on toteutettu liitteissä 1-6.

MVVM-suunnittelumallia soveltavassa projektissa uudelleenkäytettäviä näkymämalleja hyödynnettiin muun muassa komentoja määrittelevien skriptien toteutuksessa. Järjestelmään toteutettiin yksi näkymämalli, joka mahdollisti komentojen lisäämisen muokkaamisen ja hallinnan. Tätä samaa näkymämallia voitiin hyödyntää järjestelmän eri näkymissä. Kappaleessa 3.3 esitetyn esimerkkisovelluksen tavoin yksi näkymä tarjosi toiminnallisuuden komentojen muokkaamiselle, ja useampi näkymä tarjosi toiminnallisuuden komentojen esittämiselle eri tavoilla.

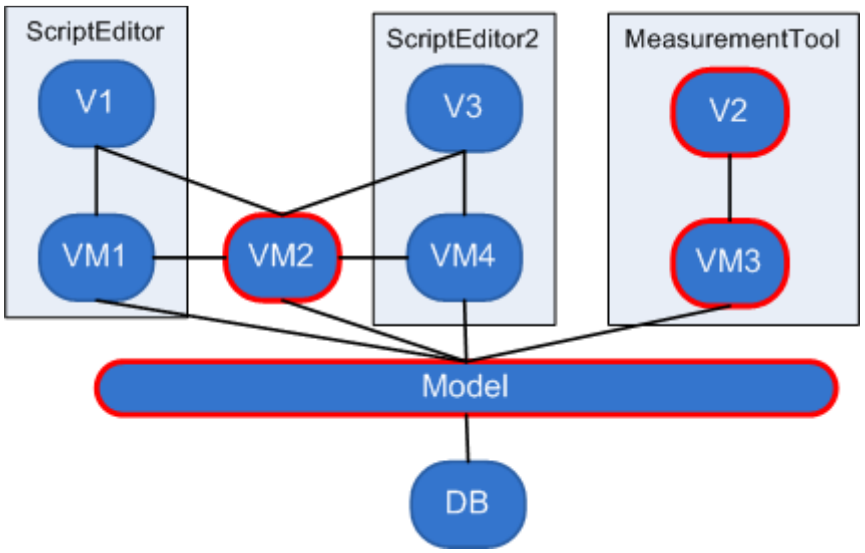
5.5 Uuden kokonaisuuden lisääminen

Atostekin projektissa toteutettu mittausjärjestelmä koostuu kahdesta kokonaisuudesta kuvan 28 mukaisesti. *ScriptEditor* toteuttaa toiminnallisuudet skriptien luomiseen ja hallintaan. *MeasurementTool* toteuttaa toiminnallisuuden mittauksien ajamiseen ja laitteiden hallintaan. *Model* kuvaa järjestelmän tietomallia ja *DB* järjestelmän tietokantaa.



Kuva 28. *Projektissa toteutetun järjestelmän ScriptEditor ja MeasurementTool kokonaisuudet.*

Tulevaisuudessa järjestelmään pitää lisätä laitteiden virrankulutusta mittaavia ominaisuuksia. Uuden kokonaisuuden tuomia muutoksia on havainnollistettu kuvassa 29.



Kuva 29. *Tulevaisuudessa lisättävä ScriptEditor2 kokonaisuus lisättynä järjestelmään.*

Kuvassa 29 punaisella kehyksellä merkatut osat voidaan käyttää uudelleen pienin muutoksin. Kuva 29 havainnollistaa, että järjestelmän tietokanta pysyy muuttumattomana. Ensimmäisessä vaiheessa varauduttiin virrankulutusmittauksien lisäämiseen, joten tietokanta on toteutettu niin, että se pitää sisällään tietokantarakenteet uudelle kokonaisuudelle. Myös vanhat laitteet mahdollistavat virrankulutusmittaukset sellaisenaan.

Virrankulutusmittaukset toteuttavat skriptit pitää luoda ja hallita uudessa kokonaisuudessa. Tätä varten kuvassa 29 on esitetty uusia näkymiä (V3) ja näkymämalleja (VM4). Järjestelmän komentoja käsittelevä VM2-näkymämalli voidaan käyttää uudelleen. Komentojen käsittely on tärkeässä asemassa mittauksien luomisessa. Suurin kokonaisuus editoreiden toteutuksessa on komentojen käsittelyä. Komentoja käsittelevä VM2-näkymämalli voidaan uudelleenkäyttää uudessa osassa lähes täysin. Näkymämalliin tulee kuitenkin lisätä virrankulutukseen liittyviä ominaisuuksia. Uusien ominaisuuksien lisääminen näkymämalliin ei vaikuta vanhan järjestelmän toimintaan. Vanhan järjestelmän näkymät on sidottu vain näkymämallin vanhoihin ominaisuuksiin.

Kuvasta 29 nähdään, että mittauksien ajamiseen tarkoitettua työkalua voidaan käyttää myös virrankulutusmittauksien suorittamiseen. Molempien mittauksien suorittaminen laitteille on komentojen lähettämistä, joten mittauksien suorittamisessa suoritetaan mallin määrittämiä komentoja eri laitteille. Virrankulutusmittaukset määrittelevät uusia komentoja, joita suoritetaan laitteilla. Tätä varten mittauksien suorittamiseen liittyviin komponentteihin tulee lisätä lisää ominaisuuksia.

5.6 Opittavuus

Projektissa saatujen havaintojen perusteella MVVM-suunnittelumalli oli nopeasti opittavissa. Projektiin lisättiin uusia työntekijöitä kesken projektin, jolloin uusille toteuttajille tuli opettaa järjestelmän rakenne ja toteutustekniikat. Opittavuutta helpotti se, että kirjallisuudessa esitetty teoria MVVM-suunnittelumallin rakenteesta toteutuvat myös käytännössä.

Microsoftin WPF-teknologia on kehitetty tukemaan MVVM-suunnittelumallia. WPF-teknologian työkalut ja menetelmät ovat hyvin dokumentoituja ja kaikkien saatavilla *Microsoftin MSDN Library*-sivuilla (MSDN 2012). Hyvän dokumentaatio ansiosta oppiminen on helpompaa. WPF-teknologian työkalut ja menetelmät ratkaisevat monia ongelmia käyttöliittymäsovelluksien toteutuksessa, jolloin toteuttajan ei tarvitse kehittää omia menetelmiä ongelmien ratkaisemiseksi. Käyttämällä WPF-teknologian menetelmiä, saavutetaan suorituskykyä, yhdenmukaisuutta ja parempaa opittavuutta käyttöliittymäsovelluksien toteutuksessa.

6 YHTEENVETO

Tämän diplomityö käsitteli malli-näkymä-arkkitehtuurin soveltamista WPF-teknologialla. Diplomityön tarkoituksena oli selvittää, kuinka kirjallisuudessa hyväksi todettu malli-näkymä-arkkitehtuuri MVVM ja käyttöliittymäkirjasto WPF ratkaisevat todellisuudessa ongelmia, jotka liittyvät käyttöliittymäsovelluksien toteuttamiseen.

Työssä havaittiin, että tunnetuimmat malli-näkymä-arkkitehtuurit MVC ja MVP tarjoavat vain korkean tason kuvauksen järjestelmän arkkitehtuurista. Korkean tason kuvaus ei tarjoa käytännön ratkaisuja toteuttajalle, jolloin arkkitehtuurimallin soveltamisesta saavutetut hyödyt riippuvat enemmän yksilön taidoista, kuin käytettävästä arkkitehtuurimallista. Lisäksi havaittiin, että MVC-arkkitehtuurimallin toteuttaminen nykyaikaisilla ohjelmointiympäristöillä on mahdotonta.

MVVM-suunnittelumallin mukaisen järjestelmän toteutusta havainnollistettiin esimerkkisovelluksen avulla. Esimerkkisovelluksen avulla käytiin läpi WPF-teknologian ominaisuuksia, jotka liittyvät MVVM-suunnittelumallin toteuttamiseen. MVVM-suunnittelumallin soveltamista käsiteltiin Atostekin projektissa saatujen havaintojen perusteella. MVVM-suunnittelumallin soveltamisessa esiteltiin projektissa tehdyt toteutusratkaisut käyttöliittymäsovelluksien yleisille toiminnallisuuksille.

Tässä työssä tehtyjen havaintojen ja Atostekin projektissa saatujen kokemuksien perusteella MVVM-suunnittelumalli mahdollistaa käyttöliittymäsovelluksen modulaarisuuden, suorituskyvyn, toteutettavuuden, ylläpidon ja opittavuuden parantamisen. MVVM-suunnittelumalli ja WPF-teknologia tarjoavat dokumentoituja ja tehokkaita työkaluja yleisimpien käyttöliittymäsovelluksien ominaisuuksien toteutukseen. Tämän ansiosta käyttöliittymäsovelluksen runko on helppo toteuttaa, jolloin aikaa jää enemmän varsinaisen sovelluslogiikan toteuttamiseen. Hyödyntämällä MVVM-suunnittelumallia voidaan saavuttaa yhdenmukaisuutta myös eri järjestelmien välillä. Tämän ansiosta yrityksessä toteutetut eri järjestelmät ovat arkkitehtuuriltaan yhdenmukaisia, jolloin eri järjestelmien yleinen rakenne on helposti opittavissa.

WPF-teknologia ja MVVM-suunnittelumalli ovat vahvasti sidoksissa toisiinsa. Tässä työssä saatujen havaintojen perusteella WPF-teknologialla toteutettava käyttöliittymäsovellus on toteutettava MVVM-suunnittelumallin mukaisesti. MVVM-suunnittelumallin mukaisen järjestelmän toteuttaminen muulla kuin WPF-teknologialla on työlästä. MVVM-suunnittelumalli hyödyntää paljon WPF-teknologiaan toteutettuja ominaisuuksia kuten tiedon sidonta, tiedon validointi ja toiminnon suorittaminen. Näiden ominaisuuksien toteuttaminen toisella ohjelmointityökalulla vaatii ylimääräistä työtä. Tämän työn perusteella ohjelmoitaessa käyttöliittymäsovelluksia *Windows-*

ympäristössä on MVVM-suunnittelumallin ja WPF-teknologian hyödyntäminen suositeltavaa.

LÄHTEET

Anderson, C. 2009. Pro Business Applications with Silverlight 4. United States of America, Apress. 500 s.

BackgroundWorker. 2011. MSDN Library. [WWW]. [Viitattu 17.02.2012]. Saatavissa: http://msdn.microsoft.com/en-us/library/system.componentmodel. backgroundworker.aspx

Bass, L., Clements, P. & Kazman R. 2003. Software Architecture in Practice. Second Edition. United States of America, Addison-Wesley Professional. 560 s.

Buschmann, F., Meunier, R., Rohnert, H., Sommerland, P. & Stal, M. 1996. Pattern-Oriented Software Architecture: A System of Patterns. England, Wiley. 457 s.

Commanding Overview. 2011. MSDN Library. [WWW]. [Viitattu 18.11.2011]. Saatavissa: http://msdn.microsoft.com/en-us/library/ms752308.aspx

Data Binding. 2011. MSDN Library. [WWW]. [Viitattu 04.10.2011]. Saatavissa: http://msdn.microsoft.com/en-us/library/ms750612.aspx

Fowler, M. 2004. Presentation Model. [WWW]. [Viitattu 09.12.2011]. Saatavissa: http://martinfowler.com/eaaDev/PresentationModel.html

Fowler, M. 2006. Development of Future Patterns of Enterprise Application Architecture. [WWW]. [Viitattu 09.05.2011]. Saatavissa: http://martinfowler.com/eaaDev/index.html

Func Delegate. 2011. MSDN Library. [WWW]. [Viitattu 17.02.2012]. Saatavissa: http://msdn.microsoft.com/en-us/library/dd269654.aspx

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. 1994. Design Patterns: Elements of Reusable Object-Oriented Software. United States of America, Addison-Wesley Professional Computing Series. 395 s.

Garofalo, R. 2011. Building Enterprise Applications with Windows® Presentation Foundation and the Model View ViewModel Pattern. United States of America, Microsoft Press. 224 s.

ICommand. 2011. MSDN Library. [WWW]. [Viitattu 18.11.2011]. Saatavissa: http://msdn.microsoft.com/en-us/library/ms616869.aspx

Interactive Application Architecture Patterns. 2007. [WWW]. [Viitattu 19.04.2011]. Saatavissa: http://www.aspiringcraftsman.com/2007/08/25/interactive-application-architecture/

Lerman, J. 2010. Programming Entity Framework. Second Edition. United States of America, O'Reilly Media. 873 s.

MacDonald, M. 2010. Pro WPF in C# 2010: Windows Presentation Foundation in .NET 4. United States of America, Apress. 1181 s.

MessageBox. 2011. MSDN Library. [WWW]. [Viitattu 17.02.2012]. Saatavissa: http://msdn.microsoft.com/en-us/library/system.windows.forms.messagebox%28v=vs.100%29.aspx

MSDN. 2012. MSDN Library. [WWW]. [Viitattu 30.03.2012]. Saatavissa: http://msdn.microsoft.com/en-us/library/default.aspx

Potel, M. 1996. Model-View-Presenter – The Taligent Programming Model for C++ and Java. [WWW]. [Viitattu 20.04.2011]. Saatavissa: http://www.wildcrest.com/Potel/Portfolio/mvp.pdf

Prism – Developer's Guide to Microsoft Prism 4.0. 2010 [WWW]. [Viitattu 06.05.2011]. Saatavissa: http://compositewpf.codeplex.com/releases/view/55580

Reenskaug, T. 2011. Trygve Reenskaug homepage. [WWW]. [Viitattu 19.04.2011]. Saatavissa: http://heim.ifi.uio.no/~trygver/

Reenskaug, T. 2007. The original MVC reports. Department of Informatics. University of Oslo. [WWW]. [Viitattu 19.04.2011]. Saatavissa: http://heim.ifi.uio.no/~trygver/2007/MVC_Originals.pdf

Smith, J. 2009. WPF Apps With The Model-View-ViewMode Design Pattern. MSDN Magazine. [WWW]. [Viitattu 16.05.2011]. Saatavissa: http://msdn.microsoft.com/en-us/magazine/dd419663.aspx

Windows Presentation Foundation. 2011. MSDN Library. [WWW]. [Viitattu 31.12.2011]. Saatavissa: http://msdn.microsoft.com/en-us/library/ms754130.aspx

XAML Overview. 2011. MSDN Library. [WWW]. [Viitattu 1.12.2011]. Saatavissa: http://msdn.microsoft.com/en-us/library/ms752059.aspx

Liite 1: AddressBookMainWindow-näkymä

```xaml
<Window x:Class="AddressBook.View.AddressBookMainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:uc="clr-namespace:AddressBook.View"
        Title="Address Book" Height="435" Width="300">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>

        <Menu Grid.Row="0">
            <MenuItem Header="_Views">
                <MenuItem Name="SearchMenuItem" Header="Search" Click="SearchMenuItem_Click"/>
                <MenuItem Name="AddMenuItem" Header="Add Contact" Click="AddMenuItem_Click"/>
            </MenuItem>
        </Menu>
        <uc:AddContactUserControl Grid.Row="1" x:Name="AddContactUserControl"></uc:AddContactUserControl>
        <uc:SearchContactUserControl Grid.Row="1" x:Name="SearchContactUserControl"></uc:SearchContactUserControl>

    </Grid>
</Window>

```

```cs
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;

namespace AddressBook.View
{
    /// <summary>
    /// Interaction logic for Window1.xaml
    /// </summary>
    public partial class AddressBookMainWindow : Window
    {
        public AddressBookMainWindow()
        {
            InitializeComponent();
            this.AddContactUserControl.Visibility = System.Windows.Visibility.Hidden;
            this.SearchContactUserControl.Visibility = System.Windows.Visibility.Visible;
        }

        private void SearchMenuItem_Click(object sender, RoutedEventArgs e)
        {
            this.AddContactUserControl.Visibility = System.Windows.Visibility.Hidden;
            this.SearchContactUserControl.Visibility = System.Windows.Visibility.Visible;
        }

        private void AddMenuItem_Click(object sender, RoutedEventArgs e)
        {
            this.AddContactUserControl.Visibility = System.Windows.Visibility.Visible;
            this.SearchContactUserControl.Visibility = System.Windows.Visibility.Hidden;
        }
    }
}
```

Liite 2: AddContactUserControl-näkymä

```
<UserControl x:Class="AddressBook.View.AddContactUserControl"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             mc:Ignorable="d"
             d:DesignHeight="402" d:DesignWidth="260" MaxWidth="260">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>

        <Grid Grid.Row="0" Grid.Column="0">
            <Grid.RowDefinitions>
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto" />
                <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>

            <Label Content="Fist Name:" Grid.Column="0" Grid.Row="0"/>
            <Label Content="Surname:" Grid.Column="0" Grid.Row="2"/>
            <Label Content="Phone:" Grid.Column="0" Grid.Row="4"/>
            <Label Content="Address:" Grid.Column="0" Grid.Row="6"/>
            <Label Content="Postcode:" Grid.Column="0" Grid.Row="7"/>
            <Label Content="City:" Grid.Column="0" Grid.Row="9"/>

            <TextBox Grid.Column="1" Grid.Row="0" x:Name="textBoxFirstName"
                     Text="{Binding Path=FirstName, ValidatesOnDataErrors=True,UpdateSourceTrigger=PropertyChanged}"/>
            <ContentPresenter Grid.Column="1" Grid.Row="1" Margin="0,0,0,6" TextBlock.Foreground="Red"
                              Content="{Binding ElementName=textBoxFirstName, Path=(Validation.Errors).CurrentItem.ErrorContent}"/>

            <TextBox Grid.Column="1" Grid.Row="2" Height="Auto" Name="textBoxSurname"
                Text="{Binding Path=Surname, ValidatesOnDataErrors=True,UpdateSourceTrigger=PropertyChanged}"/>
            <ContentPresenter Grid.Column="1" Grid.Row="3" Margin="0,0,0,6" TextBlock.Foreground="Red"
                Content="{Binding ElementName=textBoxSurname, Path=(Validation.Errors).CurrentItem.ErrorContent}"/>

            <TextBox Grid.Column="1" Grid.Row="4" Height="Auto" Name="textBoxPhone"
                Text="{Binding Path=Phone, ValidatesOnDataErrors=True,UpdateSourceTrigger=PropertyChanged}"/>
            <ContentPresenter Grid.Column="1" Grid.Row="5" Margin="0,0,0,6" TextBlock.Foreground="Red"
                Content="{Binding ElementName=textBoxPhone, Path=(Validation.Errors).CurrentItem.ErrorContent}"/>

            <TextBox Grid.Column="1" Grid.Row="6" Height="Auto" Name="textBoxAddress"
                Text="{Binding Path=Address,UpdateSourceTrigger=PropertyChanged}"/>

            <TextBox Grid.Column="1" Grid.Row="7" Height="Auto" Name="textBoxPostcode"
                Text="{Binding Path=Postcode, ValidatesOnDataErrors=True,UpdateSourceTrigger=PropertyChanged}"/>
            <ContentPresenter Grid.Column="1" Grid.Row="8" Margin="0,0,0,6" TextBlock.Foreground="Red"
                Content="{Binding ElementName=textBoxPostcode, Path=(Validation.Errors).CurrentItem.ErrorContent}"/>

            <TextBox Grid.Column="1" Grid.Row="9" Height="Auto" Name="textBoxCity"
                Text="{Binding Path=City,UpdateSourceTrigger=PropertyChanged}"/>

        </Grid>
        <Button Grid.Row="1" Margin="6,6,6,0" Content="Add Contact" Command="{Binding Path=AddContactCommand}"/>
    </Grid>
</UserControl>
```

```
using System.Windows.Controls;
using AddressBook.ViewModel;

namespace AddressBook.View
{
    /// <summary>
    /// Interaction logic for AddContactUserControl.xaml
    /// </summary>
    public partial class AddContactUserControl : UserControl
    {
        public AddContactUserControl()
        {
            InitializeComponent();
            this.DataContext = new ContactViewModel();
        }
    }
}

```

Liite 3: SearchContactUserControl-näkymä

```
<UserControl x:Class="AddressBook.View.SearchContactUserControl"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             mc:Ignorable="d"
             d:DesignHeight="444" d:DesignWidth="300">
    <UserControl.Resources>
        <DataTemplate x:Key="contactListViewDataTemplate">
            <Border BorderBrush="Blue" BorderThickness="2" Padding="5" Margin="5">
                <Grid >
                    <Grid.RowDefinitions>
                        <RowDefinition/>
                        <RowDefinition/>
                    </Grid.RowDefinitions>
                    <Grid.ColumnDefinitions>
                        <ColumnDefinition Width="*"/>
                        <ColumnDefinition Width="Auto"/>
                        <ColumnDefinition Width="Auto"/>
                        <ColumnDefinition Width="*"/>
                    </Grid.ColumnDefinitions>
                    <TextBlock Grid.Row="0" Grid.Column="0" Text="Name:"/>
                    <TextBlock Grid.Row="0" Grid.Column="1" Text="{Binding Path=Surname}" Margin="6,0,0,0" />
                    <TextBlock Grid.Row="0" Grid.Column="2" Text=", "/>
                    <TextBlock Grid.Row="0" Grid.Column="3" Text="{Binding Path=FirstName}"/>
                    <TextBlock Grid.Row="1" Grid.Column="0" Text="Phone:"/>
                    <TextBlock Grid.Row="1" Grid.Column="1" Grid.ColumnSpan="3" Text="{Binding Path=Phone}"
                        Margin="6,0,0,0" />
                </Grid>
            </Border>
        </DataTemplate>

        <Style x:Key="alternatingListViewItemStyle" TargetType="{x:Type ListViewItem}">
            <Style.Triggers>
                <Trigger Property="ItemsControl.AlternationIndex" Value="1">
                    <Setter Property="Background" Value="LightBlue"/>
                </Trigger>
                <Trigger Property="ItemsControl.AlternationIndex" Value="2">
                    <Setter Property="Background" Value="LightGray"/>
                </Trigger>
            </Style.Triggers>
        </Style>
    </UserControl.Resources>

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="Auto"/>
        </Grid.ColumnDefinitions>

        <TextBox Grid.Row="0" Grid.Column="0" Margin="6" Name="SearchTextBox"
            Text="{Binding Path=SearchKey, UpdateSourceTrigger=PropertyChanged}" />
        <Button Grid.Row="0" Grid.Column="1" Margin="6" Name="SearchButton" Content="Search" Width="100"
            Command="{Binding Path=SearchCommand}"/>

        <ListView Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="2" Margin="6" Name="ContactsListView"
            ItemContainerStyle="{StaticResource alternatingListViewItemStyle}" AlternationCount="2"
            ItemsSource="{Binding Path=ListOfSearchedContacts}" ItemTemplate="{StaticResource contactListViewDataTemplate}"
            SelectedItem="{Binding Path=SelectedContact}" HorizontalContentAlignment="Stretch"/>

        <GroupBox Grid.Row="2" Grid.Column="0" Grid.ColumnSpan="2" Margin="6" Name="InformationGroupBox" Header="Information">
            <Grid>
                <Grid.RowDefinitions>
                    <RowDefinition Height="Auto"/>
                    <RowDefinition Height="Auto"/>
                    <RowDefinition Height="Auto"/>
                    <RowDefinition Height="Auto"/>
                    <RowDefinition Height="Auto"/>
                </Grid.RowDefinitions>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="Auto"/>
                    <ColumnDefinition Width="Auto"/>
                    <ColumnDefinition Width="Auto"/>
                </Grid.ColumnDefinitions>

                <TextBlock Grid.Row="0" Grid.Column="0" Text="{Binding Path=SelectedContact.Surname}"/>
                <TextBlock Grid.Row="0" Grid.Column="1" Text=", "/>
                <TextBlock Grid.Row="0" Grid.Column="2" Text="{Binding Path=SelectedContact.FirstName}"/>

                <TextBlock Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="3"
                    Text="{Binding Path=SelectedContact.Phone}"/>
                <TextBlock Grid.Row="2" Grid.Column="0" Grid.ColumnSpan="3"
                    Text="{Binding Path=SelectedContact.Address}"/>

                <TextBlock Grid.Row="3" Grid.Column="0" Text="{Binding Path=SelectedContact.Postcode}"/>
                <TextBlock Grid.Row="3" Grid.Column="1" Text=", "/>
                <TextBlock Grid.Row="3" Grid.Column="2" Text="{Binding Path=SelectedContact.City}"/>
            </Grid>
        </GroupBox>
    </Grid>
</UserControl>
```

```
using System.Windows.Controls;
using AddressBook.ViewModel;

namespace AddressBook.View
{
    /// <summary>
    /// Interaction logic for SearchContactUserControl.xaml
    /// </summary>
    public partial class SearchContactUserControl : UserControl
    {
        public SearchContactUserControl()
        {
            InitializeComponent();

            //Don't load data context in design mode.
            if (System.ComponentModel.DesignerProperties.GetIsInDesignMode(this))
                return;

            this.DataContext = new SearchContactViewModel();
        }
    }
}

```

Liite 4: BaseViewModel-näkymämalli

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ComponentModel;
using System.Diagnostics;

namespace AddressBook.ViewModel
{
    public abstract class BaseViewModel : INotifyPropertyChanged
    {
        #region Constructor
        protected BaseViewModel(){}
        #endregion // Constructor

        #region Debugging Aides
        [Conditional("DEBUG")]
        [DebuggerStepThrough]
        public void VerifyPropertyName(string propertyName)
        {
            // Verify that the property name matches a real,
            // public, instance property on this object.
            if (TypeDescriptor.GetProperties(this)[propertyName] == null)
            {
                string msg = "Invalid property name: " + propertyName;

                if (this.ThrowOnInvalidPropertyName)
                    throw new Exception(msg);
                else
                    Debug.Fail(msg);
            }
        }

        protected virtual bool ThrowOnInvalidPropertyName { get; private set; }
        #endregion

        #region INotifyPropertyChanged Members
        public event PropertyChangedEventHandler PropertyChanged;

        protected virtual void OnPropertyChanged(string propertyName)
        {
            this.VerifyPropertyName(propertyName);

            PropertyChangedEventHandler handler = this.PropertyChanged;
            if (handler != null)
            {
                var e = new PropertyChangedEventArgs(propertyName);
                handler(this, e);
            }
        }
        #endregion
    }
}
```

Liite 5: SearchContactViewModel-näkymämalli

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows.Input;

namespace AddressBook.ViewModel
{
    public class SearchContactViewModel : BaseViewModel
    {
        public SearchContactViewModel()
        {
            ListOfSearchedContacts = ContactViewModel.CreateListOfContactViewModel(null);
        }

        private List<ContactViewModel> _searchResult;
        public List<ContactViewModel> ListOfSearchedContacts
        {
            get { return _searchResult; }
            set
            {
                _searchResult = value;
                OnPropertyChanged("ListOfSearchedContacts");
            }
        }

        private ContactViewModel _selectedContact;
        public ContactViewModel SelectedContact
        {
            get {
                return _selectedContact; }
            set
            {
                _selectedContact = value;
                OnPropertyChanged("SelectedContact");
            }
        }

        private string _searchKey;
        public string SearchKey
        {
            get
            {
             if (_searchKey != null)
                    return _searchKey;

                return "";
            }
            set
            {
              if (value != null)
                {
                    _searchKey = value.Trim();
                }
                else
                {
                    _searchKey = value;
                }
                OnPropertyChanged("SearchKey");
                this.SearchCommand.Execute(null);
            }
        }

        private RelayCommand _searchCommand;
        public ICommand SearchCommand
        {
            get
            {
                if (_searchCommand == null)
                {
                    _searchCommand = new RelayCommand(
                        param => this.ExecuteSearch()
                        );
                }
                return _searchCommand;
            }
        }

        private void ExecuteSearch()
        {
            ListOfSearchedContacts = ContactViewModel.CreateListOfContactViewModel(this.SearchKey);
        }
    }
}
```

Liite 6:ContactViewModel-näkymämalli

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using AddressBook.Model;
using AddressBook.Properties;
using System.ComponentModel;
using System.Windows.Input;

namespace AddressBook.ViewModel
{
    public class ContactViewModel : BaseViewModel , IDataErrorInfo
    {
        public static List<ContactViewModel> CreateListOfContactViewModel(string searchKey)
        {
            List<ContactViewModel> newList = new List<ContactViewModel>();
            foreach (var contact in ModelManager.GetContacts(searchKey))
            {
                newList.Add(new ContactViewModel(contact));
            }
            return newList;
        }

        private Contact _contact;
        private Contact Contact
        {
            get
            {
                return _contact;
            }
            set
            {
                _contact = value;
                base.OnPropertyChanged("FirstName");
                base.OnPropertyChanged("Surname");
                base.OnPropertyChanged("Phone");
                base.OnPropertyChanged("Address");
                base.OnPropertyChanged("Postcode");
                base.OnPropertyChanged("City");
            }
        }

        public ContactViewModel()
        {
            this.Contact = new Contact();
        }

        public ContactViewModel(Contact contactData)
        {
            this.Contact = contactData;
        }

        #region Contact Properties
        public string FirstName
        {
            get
            {
                return _contact.FirstName;
            }
            set
            {
                if (value == _contact.FirstName)
                    return;

                _contact.FirstName = value;
                base.OnPropertyChanged("FirstName");
            }
        }

        public string Surname
        {
            get
            {
                return _contact.Surname;
            }
            set
            {
                if (value == _contact.Surname)
                    return;

```

```
                _contact.Surname = value;
                base.OnPropertyChanged("Surname");
            }
        }

        public string Phone
        {
            get
            {
                return _contact.Phone;
            }
            set
            {
                if (value == _contact.Phone)
                    return;

                _contact.Phone = value;
                base.OnPropertyChanged("Phone");
            }
        }

        public string Address
        {
            get
            {
                return _contact.Address;
            }
            set
            {
                if (value == _contact.Address)
                    return;

                _contact.Address = value;
                base.OnPropertyChanged("Address");
            }
        }

        public string Postcode
        {
            get
            {
                return _contact.Postcode;
            }
            set
            {
                if (value == _contact.Postcode)
                    return;

                _contact.Postcode = value;
                base.OnPropertyChanged("Postcode");
            }
        }

        public string City
        {
            get
            {
                return _contact.City;
            }
            set
            {
                if (value == _contact.City)
                    return;

                _contact.City = value;
                base.OnPropertyChanged("City");
            }
        }
        #endregion

        private RelayCommand _addContactCommand;
        public ICommand AddContactCommand
        {
            get
            {
                if (_addContactCommand == null)
                {
                    _addContactCommand = new RelayCommand(
                        param => this.AddContact(),
                        param => this.CanAdd
                        );
                }
                return _addContactCommand;
            }
        }

        private void AddContact()
        {
            ModelManager.AddContact(this.Contact);
            this.Contact = new Contact();
            ModelManager.SaveData();
        }

        private bool CanAdd
        {
            get { return this.Contact.IsValid; }
        }

        #region IDataErrorInfo Members
        string IDataErrorInfo.Error
        {
            get {return null; }
        }

        string IDataErrorInfo.this[string propertyName]
        {
            get
            {
                string error = null;
                error = (_contact as IDataErrorInfo)[propertyName];
                return error;
            }
        }

        #endregion // IDataErrorInfo Members
    }
}
```

Liite 7: Contact-luokka

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ComponentModel;
using System.Text.RegularExpressions;

namespace AddressBook.Model
{
    public class Contact : IDataErrorInfo
    {
        public Contact(){ }

        public Contact(string firstName, string surname, string phone, string address, string postcode, string city)
        {
            FirstName = firstName;
            Surname = surname;
            Phone = phone;
            Address = address;
            Postcode = postcode;
            City = city;
        }

        private string _fristName;
        public string FirstName
        {
            get { return _fristName; }
            set { _fristName = value; }
        }

        private string _surname;
        public string Surname
        {
            get { return _surname; }
            set { _surname = value; }
        }

        private string _phone;
        public string Phone
        {
            get { return _phone; }
            set { _phone = value; }
        }

        private string _address;
        public string Address
        {
            get { return _address; }
            set { _address = value; }
        }

        private string _postcode;
        public string Postcode
        {
            get { return _postcode; }
            set { _postcode = value; }
        }

        private string _city;
        public string City
        {
            get { return _city; }
            set { _city = value; }
        }


        #region IDataErrorInfo Members

        string IDataErrorInfo.Error { get { return null; } }

        string IDataErrorInfo.this[string propertyName]
        {
            get { return this.GetValidationError(propertyName); }
        }

        #endregion // IDataErrorInfo Members

        #region Validation

        /// <summary>
```

```
81          /// Returns true if this object has no validation errors.
82          /// </summary>
83          public bool IsValid
84          {
85              get
86              {
87                  foreach (string property in ValidatedProperties)
88                      if (GetValidationError(property) != null)
89                          return false;
90
91                  return true;
92              }
93          }
94
95          static readonly string[] ValidatedProperties =
96          {
97              "FirstName",
98              "Surname",
99              "Phone",
100             "Postcode",
101         };
102
103         string GetValidationError(string propertyName)
104         {
105             if (Array.IndexOf(ValidatedProperties, propertyName) < 0)
106                 return null;
107
108             string error = null;
109
110             switch (propertyName)
111             {
112                 case "FirstName":
113                     error = this.ValidateFirstName();
114                     break;
115
116                 case "Surname":
117                     error = this.ValidateSurname();
118                     break;
119
120                 case "Phone":
121                     error = this.ValidatePhone();
122                     break;
123
124                 case "Postcode":
125                     error = this.ValidatePostcode();
126                     break;
127
128                 default:
129                     System.Diagnostics.Debug.Fail("Unexpected property being validated on Customer: " + propertyName);
130                     break;
131             }
132             return error;
133         }
134
135         string ValidateFirstName()
136         {
137             if (IsStringMissing(this.FirstName))
138                 return "First name is required!";
139
140             return null;
141         }
142
143         string ValidateSurname()
144         {
145             if (IsStringMissing(this.Surname))
146                 return "Last name is required!";
147
148             return null;
149         }
150
151         string ValidatePhone()
152         {
153             if (!IsPhoneNumber(this.Phone))
154                 return "Phone number is invalid!";
155
156             return null;
157         }
158
159         string ValidatePostcode()
160         {
```

```
            if (!IsPostcode(this.Postcode))
                return "Postcode is invalid!";

            return null;
        }

        private bool IsPostcode(string value)
        {
            if (IsStringMissing(value))
                return true;

            if (value.Length != 5)
                return false;

            Regex regex = new Regex("^[0-9]+$");
            if (regex.IsMatch(value))
                return true;

            return false;
        }

        private bool IsPhoneNumber(string value)
        {
            if (value == null || IsStringMissing(value))
                return true;

            Regex regex = new Regex("^[0-9]+$");
            if (regex.IsMatch(value))
                return true;

            return false;
        }

        static bool IsStringMissing(string value)
        {
            return String.IsNullOrEmpty(value) ||
                value.Trim() == String.Empty;
        }
        #endregion // Validation
    }
}
```

Liite 8: ModelManager-luokka

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace AddressBook.Model
{
    public class ModelManager
    {
        private static List<Contact> _listOfContacts = null;
        public static List<Contact> GetContacts(string searchKey)
        {
            List<Contact> returnContacts = new List<Contact>();

            if(_listOfContacts == null)
                _listOfContacts = XmlParser.GetContacts();

            returnContacts = _listOfContacts;


            if (searchKey != null)
            {
                returnContacts = FilterContacts(searchKey, _listOfContacts);
            }

            return returnContacts.OrderBy(c => c.Surname).ToList();
        }

        private static List<Contact> FilterContacts(string searchKey, List<Contact> listOfContacts)
        {
            string[] keys = searchKey.Split(' ');

            IEnumerable<Contact> queryContacts = listOfContacts;
            foreach (var key in keys)
            {
                var tmp =
                (from c in queryContacts
                where c.FirstName.ToUpper().Contains(key.ToUpper()) ||
                c.Surname.ToUpper().Contains(key.ToUpper()) ||
                c.Phone.ToUpper().Contains(key.ToUpper()) ||
                c.Address.ToUpper().Contains(key.ToUpper()) ||
                c.Postcode.ToUpper().Contains(key.ToUpper()) ||
                c.City.ToUpper().Contains(key.ToUpper())
                select c).ToList();

                queryContacts = tmp;
            }


            return queryContacts.ToList();
        }

        public static void AddContact(Contact contact)
        {
            _listOfContacts.Add(contact);
        }

        public static void SaveData()
        {
            XmlParser.SaveData("contacts.xml", _listOfContacts);
        }
    }
}
```

Liite 9: XmlParser-luokka

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml;

namespace AddressBook.Model
{
    public class XmlParser
    {
        public static List<Contact> GetContacts()
        {
            List<Contact> newContactList = LoadData("contacts.xml");
            return newContactList;
        }
        private static List<Contact> LoadData(string fileName)
        {
            List<Contact> listOfContacts = new List<Contact>();
            // Create an XML reader for this file.
            using (XmlReader reader = XmlReader.Create(fileName))
            {
                while (reader.Read())
                {
                    // Only detect start elements.
                    if (reader.IsStartElement())
                    {
                        // Get element name and switch on it.
                        switch (reader.Name)
                        {
                            //Detect Start element
                            case "Contacts":
                                Console.WriteLine("Start element detected!");
                                break;

                            case "Contact":
                                Contact newContact = new Contact();
                                newContact.FirstName = reader["FirstName"];
                                newContact.Surname = reader["Surname"];
                                newContact.Phone = reader["Phone"];
                                newContact.Address = reader["Address"];
                                newContact.Postcode = reader["Postcode"];
                                newContact.City = reader["City"];

                                listOfContacts.Add(newContact);
                                break;
                        }
                    }
                }
            }
            return listOfContacts;
        }

        public static bool SaveData(string fileName, List<Contact> contacts)
        {
            XmlWriterSettings settings = new XmlWriterSettings();
            settings.Indent = true;
            settings.NewLineOnAttributes = true;

            using (XmlWriter writer = XmlWriter.Create(fileName, settings))
            {
                writer.WriteStartDocument();
                writer.WriteStartElement("Contacts");

                foreach (Contact contact in contacts)
                {
                    writer.WriteStartElement("Contact");
                    writer.WriteAttributeString("FirstName", contact.FirstName);
                    writer.WriteAttributeString("Surname", contact.Surname);
                    writer.WriteAttributeString("Phone", contact.Phone);
                    writer.WriteAttributeString("Address", contact.Address);
                    writer.WriteAttributeString("Postcode", contact.Postcode);
                    writer.WriteAttributeString("City", contact.City);

                    writer.WriteEndElement();
                }
                writer.WriteEndElement();
                writer.WriteEndDocument();
            }
            return true;
        }
    }
}
```

Liite 10: RelayCommand-luokka

```
1   using System;
2
3   namespace AddressBook.ViewModel
4   {
5       public class RelayCommand : System.Windows.Input.ICommand
6       {
7           #region Fields
8           private readonly Action<object> _execute;
9           private readonly Predicate<object> _canExecute;
10          #endregion // Fields
11
12          #region Constructors
13          /// <summary>
14          /// Creates a new command that can always execute.
15          /// </summary>
16          /// <param name="execute">The execution logic.</param>
17          public RelayCommand(Action<object> execute)
18              : this(execute, null)
19          {
20          }
21
22          /// <summary>
23          /// Creates a new command.
24          /// </summary>
25          /// <param name="execute">The execution logic.</param>
26          /// <param name="canExecute">The execution status logic.</param>
27          public RelayCommand(Action<object> execute, Predicate<object> canExecute)
28          {
29              if (execute == null)
30                  throw new ArgumentNullException("execute");
31
32              _execute = execute;
33              _canExecute = canExecute;
34          }
35          #endregion // Constructors
36
37          #region ICommand Members
38          /// <summary>
39          /// Determines can the command be executed.
40          /// </summary>
41          /// <param name="parameter">Paramaters for the method</param>
42          /// <returns></returns>
43          public bool CanExecute(object parameter)
44          {
45              return _canExecute == null ? true : _canExecute(parameter);
46          }
47
48          public event EventHandler CanExecuteChanged
49          {
50              add { System.Windows.Input.CommandManager.RequerySuggested += value; }
51              remove { System.Windows.Input.CommandManager.RequerySuggested -= value; }
52          }
53
54          /// <summary>
55          /// Execute command.
56          /// </summary>
57          /// <param name="parameter">Parameters for the command.</param>
58          public void Execute(object parameter)
59          {
60              _execute(parameter);
61          }
62          #endregion // ICommand Members
63      }
64  }
```

Liite 11: ViewModelFactory-luokka

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;

namespace Examples
{
    public class ViewModelFactory
    {
        private static ViewModelFactory _instance;
        private Dictionary<Type, BaseViewModel> _viewModelRepository;
        private Func<string, string, MessageBoxButton, MessageBoxImage,
                     MessageBoxResult, MessageBoxResult> _messageBox;

        protected ViewModelFactory()
        {
            _messageBox =
                (Func<string, string, MessageBoxButton, MessageBoxImage,
                    MessageBoxResult, MessageBoxResult>)
                ((msg, capt, buttons, icon, defResult) =>
                    MessageBox.Show(msg, capt, buttons, icon, defResult));

            _viewModelRepository = new Dictionary<Type, BaseViewModel>();
        }

        public static ViewModelFactory Instance()
        {
            if (_instance == null)
                _instance = new ViewModelFactory();

            return _instance;
        }

        public BaseViewModel GetViewModelByType(Type vmType)
        {
            if (!_viewModelRepository.ContainsKey(vmType))
            {
                object[] vmArgs = new object[] { _messageBox };
                _viewModelRepository.Add(vmType,
                    (BaseViewModel)Activator.CreateInstance(vmType, vmArgs));
            }
            return _viewModelRepository[vmType];
        }

        public void UpdateAllViewModels()
        {
            foreach (var baseViewModelInstance in _viewModelRepository.Values)
            {
                baseViewModelInstance.UpdateViewModelCommand.Execute(null);
            }
        }

    }
}
```

Liite 12: BaseViewModel-näkymämalli

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Windows;
using System.Windows.Input;

namespace Examples
{
    public abstract class BaseViewModel : INotifyPropertyChanged
    {
        private RelayCommand _closeCommand;
        private RelayCommand _updateCommand;
        protected Func<string, string, MessageBoxButton, MessageBoxImage,
            MessageBoxResult, MessageBoxResult> _messageBox;
        protected List<string> _viewModelProperties;

        protected BaseViewModel(string viewModelName,
            Func<string, string, MessageBoxButton, MessageBoxImage,
            MessageBoxResult, MessageBoxResult> messageBox)
        {
            ViewModelName = viewModelName;
            _messageBox = messageBox;

            if (null == ViewModelProperties)
            {
                this.ViewModelProperties = new List<string>() { };
            }
        }

        protected List<string> ViewModelProperties
        {
            get{ return _viewModelProperties; }
            set{ _viewModelProperties = value; }
        }

        protected string ViewModelName { get; set; }

        public ICommand UpdateViewModelCommand
        {
            get
            {
                if (_updateCommand == null)
                {
                    _updateCommand = new RelayCommand(
                        param => this.UpdateViewModel()
                        );
                }
                return _updateCommand;
            }
        }

        private void UpdateViewModel()
        {
            foreach (var propertyName in _viewModelProperties)
            {
                OnPropertyChanged(propertyName);
            }
        }
```

```
59
60          public System.Windows.Input.ICommand CloseCommand
61          {
62              get
63              {
64                  if (_closeCommand == null)
65                  {
66                      _closeCommand = new RelayCommand(
67                         param => Close() );
68                  }
69                  return _closeCommand;
70              }
71          }
72
73          public event Action RequestClose;
74
75          public virtual void Close()
76          {
77              if (RequestClose != null)
78                  RequestClose();
79          }
80
81          public event PropertyChangedEventHandler PropertyChanged;
82
83          protected virtual void OnPropertyChanged(string propertyName)
84          {
85              PropertyChangedEventHandler handler = this.PropertyChanged;
86              if (handler != null)
87              {
88                  var e = new PropertyChangedEventArgs(propertyName);
89                  handler(this, e);
90              }
91          }
92      }
93  }
```

Liite 13: ProgressWindow-luokka

```
<Window x:Class="Examples.ProgressWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="" Height="140" Width="341" Loaded="Window_Loaded" MaxHeight="140" MaxWidth="341">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="Auto"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="40"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>

        <TextBlock Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="2" Margin="6,6,6,0"
                   Name="titleTextBlock" Text="Title:"/>
        <ProgressBar Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="2" Margin="6,6,6,6"
                     Name="progressBar" Value="{Binding Path=ProgressPercentageValue}" />
        <Label  Grid.Row="2" Grid.Column="0" Margin="6,6,6,6" Content="{Binding Path=ProgressStatus}"/>
        <Button Grid.Row="2" Grid.Column="1" Margin="0,6,6,6" Width="80"
                Name="cancelButton" Content="Cancel" Command="{Binding Path=CancelProgressCommand}"/>
    </Grid>
</Window>
```

```
using System;
using System.Windows;

namespace Examples
{
    public partial class ProgressWindow : System.Windows.Window
    {
        /// <summary>
        /// These lines will hide window's close,
        /// resize and window icon buttons from title bar of the window.
        /// </summary>
        private const int GWL_STYLE = -16;
        private const int WS_SYSMENU = 0x80000;
        [System.Runtime.InteropServices.DllImport("user32.dll", SetLastError = true)]
        private static extern int GetWindowLong(IntPtr hWnd, int nIndex);
        [System.Runtime.InteropServices.DllImport("user32.dll")]
        private static extern int SetWindowLong(IntPtr hWnd, int nIndex, int dwNewLong);

        private void Window_Loaded(object sender, RoutedEventArgs e)
        {

            var hwnd = new System.Windows.Interop.WindowInteropHelper(this).Handle;
            SetWindowLong(hwnd, GWL_STYLE, GetWindowLong(hwnd, GWL_STYLE) & ~WS_SYSMENU);

        }

        public ProgressWindow()
        {
            InitializeComponent();
        }

        public string TitleText
        {
            set
            {
                this.titleTextBlock.Text = value;
            }
        }

        public bool IsIndeterminate
        {
            set
            {
                this.progressBar.IsIndeterminate = value;
            }
        }

        public void HideCancelButton()
        {
            this.cancelButton.Visibility = System.Windows.Visibility.Hidden;
        }
    }
}
```

Liite 14: Tyyppimuunnokset

```
using System;
using System.Windows.Data;
using System.Windows;
using System.Collections;

namespace Examples.ValueConverter
{
    [ValueConversion(typeof(byte), typeof(string))]
    public class ByteToHexConverter : IValueConverter
    {
        object IValueConverter.Convert(object value, Type targetType,
            object parameter, System.Globalization.CultureInfo culture)
        {
            return ((byte)value).ToString("X2");
        }

        object IValueConverter.ConvertBack(object value, Type targetType,
            object parameter, System.Globalization.CultureInfo culture)
        {
            byte retval = 0x00;
            bool valid = Byte.TryParse((string)value,
                System.Globalization.NumberStyles.AllowHexSpecifier, null,
                out retval);

            if (valid)
                return retval;
            else
                return DependencyProperty.UnsetValue;
        }
    }

    [ValueConversion(typeof(byte), typeof(string))]
    public class ByteToBinConverter : IValueConverter
    {
        object IValueConverter.Convert(object value, Type targetType,
            object parameter, System.Globalization.CultureInfo culture)
        {
            byte byteValue = (byte)value;
            string retval;

            retval = ValueParsers.ByteToBinaryString(byteValue);

            return retval;
        }

        object IValueConverter.ConvertBack(object value, Type targetType,
            object parameter, System.Globalization.CultureInfo culture)
        {
            byte retval;
            bool valid = false;

            retval = ValueParsers.BinaryStringToByte((string)value, out valid);

            if (valid)
                return retval;
            else
                return null;
        }
    }

```

```
61      public static class ValueParsers
62      {
63          public static byte BinaryStringToByte(string data, out bool valid)
64          {
65              data = data.Replace(" ", "");
66
67              if (data.Length == 8)
68              {
69                  BitArray bits = new BitArray(8);
70                  for (int i = 0; i < 8; i++)
71                  {
72                      if (data[i] == '1')
73                      {
74                          bits[7-i] = true;
75                      }
76                      else if (data[i] == '0')
77                      {
78                          bits[7-i] = false;
79                      }
80                      else
81                      {
82                          valid = false;
83                          return 0x00;
84                      }
85                  }
86
87                  byte[] bytes = new byte[1];
88                  bits.CopyTo(bytes, 0);
89
90                  valid = true;
91                  return bytes[0];
92              }
93              else
94              {
95                  valid = false;
96                  return 0x00;
97              }
98          }
99
100         public static string ByteToBinaryString(byte data)
101         {
102             string returnValue = "";
103             BitArray bits = new BitArray(new byte[] { data });
104             int count = 1;
105
106             for (int i = bits.Length-1; i >= 0; i--)
107             {
108                 returnValue += bits[i] ? "1" : "0";
109                 if (count == 4)
110                     returnValue += " ";
111
112                 count++;
113             }
114             return returnValue;
115         }
116     }
117 }
118
```

Liite 15: DataGrid-komponentti

```
<Window x:Class="Examples.DataGrid"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:commonUtils="clr-namespace:Examples.CommonUtils;assembly=CommonUtils"
        Title="DataGrid Example">

    <Window.Resources>
        <DataTemplate x:Key="NotSelectedTemplate">
            <Grid>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="*"/>
                </Grid.ColumnDefinitions>
                <Label Content="{Binding Path=ParameterInfo}" Margin="0,-5,0,0" />
            </Grid>
        </DataTemplate>

        <ControlTemplate x:Key="ReadOnlyTemplate">
            <Grid>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="*"/>
                </Grid.ColumnDefinitions>
                <Label Content="{Binding Path=ParameterInfo}" Margin="0,-5,0,0" />
            </Grid>
        </ControlTemplate>

        <ControlTemplate x:Key="OnOffTemplate">
            <ComboBox Margin="2" SelectedIndex="{Binding Path=OnOffSelection}">
                <ComboBoxItem Content="On" />
                <ComboBoxItem Content="Off" />
            </ComboBox>
        </ControlTemplate>

        <ControlTemplate x:Key="StartEndTemplate">
            <Grid>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="Auto"/>
                    <ColumnDefinition Width="*"/>
                </Grid.ColumnDefinitions>
                <Grid.RowDefinitions>
                    <RowDefinition Height="Auto"/>
                    <RowDefinition Height="Auto"/>
                    <RowDefinition Height="Auto"/>
                </Grid.RowDefinitions>

                <Button Grid.Column="1" Grid.Row="0" Content="Default" MaxWidth="60"
                        HorizontalAlignment="Left" Margin="6,0,0,0"
                        Command="{Binding Path=SetStartEndToDefaultCommand}"/>
                <Label Grid.Column="0" Grid.Row="1" Content="Start:"/>
                <TextBox Grid.Column="1" Grid.Row="1" Text="{Binding Path=StartInt}"/>
                <Label Grid.Column="0" Grid.Row="2" Content="End:"/>
                <TextBox Grid.Column="1" Grid.Row="2" Text="{Binding Path=EndInt}"/>
            </Grid>
        </ControlTemplate>

        <DataTemplate x:Key="ParameterEditingDataTemplate">
            <Control x:Name="parameterControl"
                     Template="{StaticResource ReadOnlyTemplate}" />
            <DataTemplate.Triggers>

                <DataTrigger Binding="{Binding ParameterType}">
                    <DataTrigger.Value>
                        <commonUtils:ParameterType>OnOff</commonUtils:ParameterType>
                    </DataTrigger.Value>
                    <Setter TargetName="parameterControl" Property="Template"
                            Value="{StaticResource OnOffTemplate}" />
                </DataTrigger>

                <DataTrigger Binding="{Binding ParameterType}">
                    <DataTrigger.Value>
                        <commonUtils:ParameterType>StartEnd</commonUtils:ParameterType>
                    </DataTrigger.Value>
                    <Setter TargetName="parameterControl" Property="Template"
                            Value="{StaticResource StartEndTemplate}" />
                </DataTrigger>

            </DataTemplate.Triggers>
        </DataTemplate>
    </Window.Resources>

```

```
80      <Grid>
81          <DataGrid Margin="6" ItemsSource="{Binding Path=CommandScriptSteps}">
82
83              <DataGrid.Columns>
84                  <DataGridTextColumn Header="#" Binding="{Binding Index}"/>
85
86                  <DataGridTemplateColumn Header="Command">
87                      <DataGridTemplateColumn.CellTemplate>
88                          <DataTemplate>
89                              <TextBlock Text="{Binding Command}"/>
90                          </DataTemplate>
91                      </DataGridTemplateColumn.CellTemplate>
92                      <DataGridTemplateColumn.CellEditingTemplate>
93                          <DataTemplate>
94                              <ComboBox ItemsSource="{Binding Path=CommandList}"
95                                        SelectedItem="{Binding Path=Command}"/>
96                          </DataTemplate>
97                      </DataGridTemplateColumn.CellEditingTemplate>
98                  </DataGridTemplateColumn>
99
100                 <DataGridTemplateColumn Header="Parameter"
101                    CellTemplate="{StaticResource NotSelectedTemplate}"
102                    CellEditingTemplate="{StaticResource ParameterEditingDataTemplate}" >
103                 </DataGridTemplateColumn>
104
105                 <DataGridTemplateColumn Header="Details">
106                     <DataGridTemplateColumn.CellTemplate>
107                         <DataTemplate>
108                             <Button Command="{Binding Path=OpenAdvancedCommand}" Content="Details"/>
109                         </DataTemplate>
110                     </DataGridTemplateColumn.CellTemplate>
111                 </DataGridTemplateColumn>
112
113                 <DataGridTextColumn Header="Notes" Binding="{Binding Notes}"/>
114             </DataGrid.Columns>
115         </DataGrid>
116
117         </Grid>
118 </Window>
119
```